Document Number: DSP0004

Date: 2012-12-13

Version: 3.0.0

Common Information Model (CIM) Metamodel

Document Type: Specification

Document Status: DMTF Standard

Document Language: en-US

CONTENTS

Figures

Tables

Foreword

The Common Information Model (CIM) Metamodel (DSP0004) was prepared by the DMTF Architecture Working Group.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. For information about the DMTF, see http://www.dmtf.org.

Acknowledgments

The DMTF acknowledges the following individuals for their contributions to this document:

Editor:

- George Ericson – EMC

Contributors:

- Andreas Maier – IBM
- Jim Davis – WBEM Solutions
- Karl Schopmeyer – Inova Development
- Lawrence Lamers – VMware
- Wojtek Kozaczynski – Microsoft

Introduction

This document specifies the DMTF Common Information Model (CIM) Metamodel. The role of CIM Metamodel is to define the semantics for the construction of conformant models and the schema that represents those models.

The primary goal of specifying the CIM Metamodel is to enable sharing of elements across independently developed models for the construction of new models and interfaces.

Modeling requirements and environments are often different and change over time. The metamodel is further enhanced with the capability of extending its elements through the use of qualifiers.

The Common Information Model (CIM) schema published by DMTF is a schema that is conformant with the CIM Metamodel. The CIM is a rich and detailed ontology for computer and systems management.

The CIM Metamodel is based on a subset of the UML metamodel (as defined in the Unified Modeling Language: Superstructure specification) with the intention that elements that are modeled in a UML user model can be incorporated into a CIM schema with little or no modification.

In addition, any CIM schema can be represented as a UML user model, enabling the use of commonly available UML tools to create and manage CIM schema.

Document conventions

Typographical conventions

The following typographical conventions are used in this document:

- Document titles are marked in *italics*.
- Important terms that are used for the first time are marked in *italics*.
- ABNF rules and OCL text are in monospaced font.

ABNF usage conventions

Format definitions in this document are specified using ABNF (see RFC5234), with the following deviations:

- Literal strings are to be interpreted as case-sensitive UCS/Unicode characters, as opposed to the definition in RFC5234 that interprets literal strings as case-insensitive US-ASCII characters.
- In previous versions of this document, the vertical bar (|) was used to indicate a choice. Starting with version 2.6 of this document, the forward slash (/) is used to indicate a choice, as defined in RFC5234.

Naming conventions

Upper camel case is used at all levels for the names of model or metamodel elements (e.g., Element, TypedElement or ComplexValue). Lower camel case is used for the names of attributes of model or metamodel elements (e.g., value and defaultValue).

Deprecated material

Deprecated material is not recommended for use in new development efforts. Existing and new implementations may rely on deprecated material, but should move to the favored approach as soon as possible. Implementations that are conformant to this specification shall implement any deprecated elements as required by this document in order to achieve backwards compatibility.

The following typographical convention indicates deprecated material:

DEPRECATED

Deprecated material appears here.

DEPRECATED

In places where this typographical convention cannot be used (for example, tables or figures), the "DEPRECATED" label is used alone.

Experimental material

Experimental material has yet to receive sufficient review to satisfy the adoption requirements set forth by the DMTF. Experimental material is included in this document as an aid to implementers of implementations conformant to this specification who are interested in likely future developments. Experimental material may change as implementation experience is gained. It is likely that experimental material will be included in an upcoming revision of the document. Until that time, experimental material is purely informational.

The following typographical convention indicates experimental material:

EXPERIMENTAL

Experimental material appears here.

EXPERIMENTAL

In places where this typographical convention cannot be used (for example, tables or figures), the "EXPERIMENTAL" label is used alone.

Common Information Model (CIM) Metamodel

1 Scope

This document describes the Common Information Model (CIM) Metamodel, which is based on the Unified Modeling Language: Superstructure specification. CIM schemas represent object-oriented models that can be used to represent the resources of a managed system, including their attributes, behaviors, and relationships. The CIM Metamodel includes expressions for common elements that must be clearly presented to management applications (for example, classes, properties, methods, and associations).

This document does not describe CIM schemas or languages, related schema implementations, application programming interfaces (APIs), or communication protocols.

Provisions, (i.e. shall, should, may…), target consumers of the CIM metamodel, for example CIM schema developers.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies. For references without a date or version, the latest published edition of the referenced document (including any corrigenda or DMTF update versions) applies.

ANSI/IEEE 754-2008, IEEE® Standard for Floating-Point Arithmetic, August 29 2008
http://ieeexplore.ieee.org/servlet/opac?punumber=4610933

EIA-310, Cabinets, Racks, Panels, and Associated Equipment
http://global.ihs.com/doc_detail.cfm?currency_code=USD&customer_id=21254B2B350A&oshid=21254B2B350A&shopping_cart_id=21254B2B350A&rid=Z56&mid=5280&country_code=US&lang_code=ENGL&item_s_key=00032880&item_key_date=940031&input_doc_number=&input_doc_title=Cabinets%2C%20Racks%2C%20Panels

IETF RFC3986, Uniform Resource Identifiers (URI): Generic Syntax, August 1998
http://tools.ietf.org/html/rfc3986

IETF RFC5234, Augmented BNF for Syntax Specifications: ABNF, January 2008
http://tools.ietf.org/html/rfc5234

ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards
http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype

ISO 1000:1992, SI units and recommendations for the use of their multiples and of certain other units
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=5448

ISO 8601:2004 (E), Data elements and interchange formats – Information interchange — Representation of dates and times
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40874

IEC 80000-13:2008, Quantities and units - Part 13: Information science and technology,
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=31898

ISO/IEC 10646:2012, Information technology — Universal Coded Character Set (UCS)
http://standards.iso.org/ittf/PubliclyAvailableStandards/c056921_ISO_IEC_10646_2012.zip

OMG, Object Constraint Language, Version 2.3.1
http://www.omg.org/spec/OCL/2.3.1

OMG, Unified Modeling Language: Superstructure, Version 2.3
http://www.omg.org/spec/UML/2.3/Superstructure/PDF/

The Unicode Consortium, Unicode 6.1.0, Unicode Standard Annex #15: Unicode Normalization Forms
http://www.unicode.org/reports/tr15/tr15-35.html

W3C, *Character Model for the World Wide Web 1.0: Normalization*, Working Draft, 27 October 2005,
http://www.w3.org/TR/charmod-norm/

W3C, NamingContexts in XML, W3C Recommendation, 14 January 1999,
http://www.w3.org/TR/REC-xml-names

3 Terms and definitions

In this document, some terms have a specific meaning beyond the normal English meaning. Those terms are defined in this clause.

The terms "shall" ("required"), "shall not", "should" ("recommended"), "should not" ("not recommended"), "may", "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Annex H. The terms in parenthesis are alternatives for the preceding term, for use in exceptional cases when the preceding term cannot be used for linguistic reasons. ISO/IEC Directives, Part 2, Annex H specifies additional alternatives. Occurrences of such additional alternatives shall be interpreted in their normal English meaning.

The terms "clause", "subclause", "paragraph", and "annex" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 5.

The terms "normative" and "informative" in this document are to be interpreted as described in ISO/IEC Directives, Part 2, Clause 3. In this document, clauses, subclauses, or annexes labeled "(informative)" do not contain normative content. Notes and examples are always informative elements.

The following additional terms are used in this document.

3.1

Cardinality

the number of elements

3.2

CIM Metamodel

the metamodel described in this document, defining the semantics for the construction of schemas that conform to the metamodel

3.3

CIM schema

a formal language representation of a model, (including but not limited to CIM Schema), that is conformant to the CIM Metamodel

3.4

CIM Schema

the CIM schema with schema name "CIM" that is published by DMTF. The CIM Schema defines an ontology for management.

3.5
conformant
in agreement with the requirements and constraints of a specification

3.6
implementation
a realization of a model or metamodel

3.7
instance
the run-time realization of a class from a model

3.8
key
key property
a property whose value uniquely identifies an instance within some scope of uniqueness

3.9
model
set of entities and the relationships between them that define the semantics, behavior and state of that set

3.10
managed resource
a resource in the managed environment

NOTE This was called "managed object" in CIM v2.

3.11
multiplicity
the allowable range for the number of instances associated to an instance

3.12
Null

a state of a typed element that indicates the absence of value

3.13
subclass
a specialized class

3.14
subtype
a specialized type

3.15
superclass
a generalization of a class (i.e., a more general class)

3.16
supertype
a generalization of a type (i.e., a more general type)

3.17
Unified Modeling Language
a modeling language defined by the Unified Modeling Language (UML)

4 Symbols and abbreviated terms

The following abbreviations are used in this document.

4.1
CIM
Common Information Model

4.2
OMG
Object Management Group (see: http://www.omg.org)

4.3
OCL
Object Constraint Language

4.4
UML
Unified Modeling Language

5 CIM schema elements

5.1 Introduction

This clause is targeted at developers of CIM schemas and normatively defines the elements used in their construction. The elements defined in this clause are conformant with the requirements of the CIM Metamodel (see clause 6), but this clause does not define all constraints on these elements.

5.2 Modeling a management domain

Managed resources are modeled as classes.

State of a resource is modeled as properties of a class.

Behaviors of a resource are modeled as methods of a class.

Relationships between resources are modeled as associations.

5.3 Models and schema

A model is a conceptual representation of something and a schema is a formal representation of a model, with the elements of a schema representing the essential concepts of the model.

Each schema provides a naming context for the declaration of schema elements.

The name of a schema should be globally unique across all schemas (in the world). To help achieve this goal, the schema name should include a copyrighted, trademarked or otherwise unique name that is owned by the business entity defining the schema, or is a registered ID that is assigned to that business entity by a recognized global authority. However, given that there is no central registry of schema names, this naming mechanism does not necessarily guarantee uniqueness of schema names.

The CIM Schema published by DMTF is an example of a particular schema that conforms to the CIM Metamodel.

Each schema has a version that contains monotonically increasing major, minor, and update version numbers.

5.4 Common attributes of typed elements

Certain of the model elements are not types themselves, but have a type. These elements are: properties (including references), method return values and parameters, and qualifier types. Unless otherwise restricted, any type may be used for these elements.

Collectively, elements that hold values of a type are referred to as typed elements.

Each typed element specifies whether it is intended to be accessed as an array or a scalar. The elements of an array each have the specified type.

5.4.1 Scalar

If a typed element is a scalar (i.e., not an array), it can have at most one value, and may be required to have a value (for more information on the required qualifier, see 7.25). The default is that a value is not required. Table 1 defines distinguishable states of a scalar element.

Table 1 – Distinguishable states of a scalar element

Value	Element Represented	Value Specification	Description
Not present	No	No	The element is not represented and shall be assumed to have no value unless otherwise specified.
Null	Yes	No	The element is specified with no value.
x	Yes	Yes	The value is x.

5.4.2 Array

If the array is required, it shall have a value (for more information on the required qualifier, see 7.25). If the array is not required, it may have no value (e.g., Null). If an array has a value, it contains a consecutive sequence of zero or more elements.

If an array element is present, it shall either have a value consistent with its type or have no value.

The size of an implemented, non-Null array is the count of the number of elements. Indexes into the sequence of elements start at zero and are monotonically increasing by one. (In other words, there are no gaps.) Each element has a value of the type specified by the array or is Null.

Table 2 defines distinguishable states of an array. The states depend on whether or not the array element is represented and if so, on the values of elements of the array.

Table 2 – Distinguishable states of an array element

Value	Element Represented	Values Specified	Description
N.A.	No	No	The array element is not represented and shall be assumed to have no value unless otherwise specified.
Null	Yes	No	The array is specified with no value.
[ ]	Yes	No	The array has no elements.
[Null]	Yes	Yes	The array has one element specified with no value.
[""]	Yes	Yes	The array has one element specified with an empty string value.
["x", Null, "y" …]	Yes	Yes	The array has multiple elements, some may be specified with no value.

An array shall also specify the type of array. The array type is specified by the ArrayType qualifier (see 7.3) and by the ArrayKind enumeration (see Table 3).

Table 3 – ArrayKind enumeration

Enumeration value	Description
bag	The set of element values may contain duplicates, the order of elements is not preserved, and elements may be removed or added. (Equivalent to OCL::BagType.)
set	The set of element values shall not contain duplicates, the order of elements is not preserved, and elements may be removed or added. (Equivalent to OCL::SetType.)
ordered	The set of element values may contain duplicates and elements may be removed or added. Except on element addition, removal, or on element value change, the order of elements is preserved.
orderedSet	The set of element values shall not contain duplicates and elements may be removed or added. The order of elements is preserved, except on element addition, removal, or on element value change. (Equivalent to OCL::OrderedSetType.)
indexed	The set of element values may contain duplicates, the order of elements is preserved, and individual elements shall not be removed or added. (Equivalent to OCL::SequenceType.)

5.5 Primitive types

Primitive types are predefined by the CIM Metamodel and cannot be extended at the model level. Future minor versions of this document will not add new primitive types.

NOTE Primitive types were termed "intrinsic types" in version 2 of this document.

Languages that conform to the CIM Metamodel shall support all primitive types defined in this subclause.

Table 4 lists the primitive types and describes the value space of each type. Types marked as abstract cannot be used for defining elements in CIM schemas. Their purpose is to be used in constraints that apply to all concrete types derived directly or indirectly from them.

There is no type coercion of values between these types. For example, if a CIM method has an input parameter of type uint32, the value provided for this parameter when invoking the method needs to be of type uint32.

Table 4 – Primitive types

Type Name	Abstract	Supertype	Meaning and Value Space
boolean	No		a boolean. Value space: True, False
datetime	No		a timestamp or interval in CIM datetime format. For details, see 5.5.1.
integer	Yes	*numeric*	an abstract base type for any positive or negative whole number.
numeric	Yes		an abstract base type for any numbers.
octetstring	No		a sequence octets representing the value having an arbitrary length from zero to a CIM Metamodel implementation-defined maximum. For details see 5.5.2.
real	Yes	*numeric*	an abstract base type for any IEEE-754 floating point number.
real32	No	*real*	a floating-point number in IEEE-754 Single format.
real64	No	*real*	a floating-point number in IEEE-754 Double format.
signedInteger	Yes	*integer*	an abstract base type for signed whole numbers.
sint8	No	*signedInteger*	a signed 8-bit integer. Value space: -2^7 … 2^7 - 1
sint16	No	*signedInteger*	a signed 16-bit integer. Value space: -2^15 … 2^15 - 1
sint32	No	*signedInteger*	a signed 32-bit integer. Value space: -2^31 … 2^31 - 1
sint64	No	*signedInteger*	a signed 64-bit integer. Value space: -2^63 … 2^63 - 1
string	No		a sequence of UCS characters with arbitrary length from zero to a CIM Metamodel implementation-defined maximum. For details see 5.5.3.
uint8	No	*unsignedInteger*	an unsigned 8-bit integer. Value space: 0 … 2^8 - 1
uint16	No	*unsignedInteger*	an unsigned 16-bit integer. Value space: 0 … 2^16 - 1
uint32	No	*unsignedInteger*	an unsigned 32-bit integer. Value space: 0 … 2^32 - 1
uint64	No	*unsignedInteger*	an unsigned 64-bit integer. Value space: 0 … 2^64 - 1
unsignedInteger	Yes	*integer*	an abstract base type for unsigned whole numbers.

5.5.1 **Datetime**

Values of type datetime are timestamps or intervals. If the value is representing a timestamp, it specifies a point in time in the Gregorian calendar, including time zone information, with varying precision up to microseconds. If the value is representing an interval, it specifies an amount of time, with varying precision up to microseconds.

5.5.1.1 Datetime timestamp format

Datetime is based on the proleptic Gregorian calendar, as defined in "The Gregorian calendar", which is section 3.2.1 of ISO 8601.

Note Timestamp values defined here do not have the same formats as their equivalents in ISO 8601.

Because timestamp values contain the UTC offset, the same point in time can be specified using different UTC offsets by adjusting the hour and minute fields accordingly. The UTC offset shall be preserved.

For example, Monday, May 25, 1998, at 1:30:15 PM EST is represented in datetime timestamp format `19980525133015.0000000-300`.

The year 1BC is represented as year 0000 and 0001 representing 1AD.

Values of type datetime have a fixed-size string-based format using US-ASCII characters.

The format for timestamp values is:

```
yyyymmddhhmmss.mmmmmmsutc
```

The meaning of each field is as follows:

- `yyyy` is a four-digit year.
- `mm` is the month within the year (starting with 01).
- `dd` is the day within the month (starting with 01).
- `hh` is the hour within the day (24-hour clock, starting with 00).
- `mm` is the minute within the hour (starting with 00).
- `ss` is the second within the minute (starting with 00).
- `mmmmmm` is the microsecond within the second (starting with 000000).
- `s` is a + (plus) or – (minus), indicating that the value is a timestamp, and indicating the direction of the offset from Universal Coordinated Time (UTC). A + (plus) is used for time zones east of the Greenwich meridian, and a – (minus) is used for time zones west of the Greenwich meridian.
- `utc` is the offset from UTC, expressed in minutes.

Values of a datetime timestamp formatted field shall be zero-padded so that the entire string is always 25 characters in length.

Datetime timestamp fields that are not significant shall be replaced with the asterisk (`*`) character. Fields that are not significant are beyond the resolution of the data source. These fields indicate the precision of the value and can be used only for an adjacent set of fields, starting with the least significant field (`mmmmmm`) and continuing to more significant fields. The granularity for asterisks is always the entire field, except for the `mmmmmm` field, for which the granularity is single digits. The UTC offset field shall not contain asterisks.

5.5.1.2 Datetime interval format

NOTE Interval is equivalent to the term "duration" in ISO 8601. Interval values defined here do not have the same formats as their equivalents in ISO 8601.

The format for intervals is:

```
ddddddddhhmmss.mmmmmm:000
```

The meaning of each field is:

- `dddddddd` is the number of days.
- `hh` is the remaining number of hours.
- `mm` is the remaining number of minutes.
- `ss` is the remaining number of seconds.
- `mmmmmm` is the remaining number of microseconds.
- `:` (colon) indicates that the value is an interval.
- `000` (the UTC offset field) is always zero for interval values.

For example, an interval of 1 day, 13 hours, 23 minutes, 12 seconds, and 0 microseconds would be represented as follows:

```
00000001132312.000000:000
```

Datetime interval field values shall be zero-padded so that the entire string is always 25 characters in length.

Datetime interval fields that are not significant shall be replaced with the asterisk (`*`) character. Fields that are not significant are beyond the resolution of the data source. These fields indicate the precision of the value and can be used only for an adjacent set of fields, starting with the least significant field (`mmmmmm`) and continuing to more significant fields. The granularity for asterisks is always the entire field, except for the `mmmmmm` field, for which the granularity is single digits. The UTC offset field shall not contain asterisks.

For example, if an interval of 1 day, 13 hours, 23 minutes, 12 seconds, and 125 milliseconds is measured with a precision of 1 millisecond, the format is: `00000001132312.125***:000`.

An interval value is valid if the value of each single field is in the valid range. Valid values shall not be rejected by any validity checking.

5.5.2 OctetString

The value of an octet string is represented as a sequence of zero or more octets (8-bit bytes).

An element of type octet string that is Null is distinguishable from the same element having a zero-length value, (i.e. the empty string).

5.5.3 String

Values of type string are sequences of zero or more UCS characters with the exception of UCS character U+0000. The UCS character U+0000 is excluded to permit implementations to use it within an internal representation as a string termination character.

The semantics depends on its use. It can be a comment, computational language expression, OCL expression, etc. It is used as a type for string properties and expressions.

An element of type string that is Null is distinguishable from the same element having a zero-length value, (i.e. the empty string).

For string-typed values, CIM Metamodel implementations shall support the character repertoire defined by ISO/IEC 10646. (This is also the character repertoire defined by the Unicode Standard.)

The UCS character repertoire evolves over time; therefore, it is recommended that CIM Metamodel implementations support the latest published UCS character repertoire in a timely manner.

UCS characters in string-typed values should be represented in Normalization Form C (NFC), as defined in The Unicode Standard, Annex #15: Unicode Normalization Forms. UCS characters in string-typed values shall be represented in a coded representation form that satisfies the requirements for the character repertoire stated in this subclause. Other specifications are expected to specify additional rules on the usage of particular coded representation forms (see DSP0200 as an example). In order to minimize the need for any conversions between different coded representation forms, it is recommended that such other specifications mandate the UTF-8 coded representation form (defined in ISO/IEC 10646).

See ANNEX B for a summary on UCS characters.

5.5.4 **Null**

Null is a state of a typed element that indicates the absence of value. Unless otherwise restricted any typed element may be Null.

5.6 Schema elements

5.6.1 **Enumeration**

An enumeration is a type with a literal type of string or integer and may have zero or more qualifiers (see 5.6.12). It describes a set of zero or more named values. Each named value is known as an enumeration value and has the literal type of the enumeration.

An enumeration may be defined at the schema level with a schema unique name or within a structure, (including class and association), with a structure unique name. The name of an enumeration is used as its type name.

An enumeration may directly inherit from one other enumeration. The literal type of a derived enumeration shall be the literal type of the base enumeration.

In an inheritance relationship between enumerations, the more general enumeration is called the *supertype*, and the more specialized enumeration is called the *subtype*.

A derived enumeration inherits all enumeration values exposed by its supertype enumeration (if any). These inherited enumeration values add to the enumeration values defined within the derived enumeration. The combined set of enumeration values defined and inherited is called the set of enumeration values *exposed* by the derived enumeration. There is no concept of overriding enumeration values in derived enumerations (as there is for properties of structures).

An enumeration that exposes zero enumeration values shall be abstract.

The names of all exposed enumeration values shall be unique within the defining enumeration. The following ABNF defines the syntax for local and schema level enumeration names.

```
localEnumerationName = IDENTIFIER
```

```
enumerationName = schemaName "_" IDENTIFIER
```

5.6.2 EnumValue

An enumeration value is a named value of an enumeration and may have zero or more qualifiers (see 5.6.12). If a value is not specified for an enumeration with a literal type of string, the value shall be set to the name of the enumeration value. A value shall be specified for an enumeration with a literal type of integer. The following ABNF defines the syntax for enumeration value names.

```
EnumValueName = IDENTIFIER
```

5.6.3 Property

A property is a named and typed structural feature of a structure, (including class and association). Properties may be scalars or arrays and may have zero or more qualifiers (see 5.6.12).

A property shall have a unique name within the properties of its defining type, including any inherited properties. The following ABNF defines the syntax for property names.

```
propertyName = IDENTIFIER
```

A property declaration may define a default value.

5.6.3.1 Key property

A property may be designated as a key. Each such property shall be a scalar primitive type (see 5.5) and shall not be Null.

Properties designated as containing an embedded object (see 7.9) shall not be designated as key.

5.6.3.2 Property attributes

Accessibility to a property's values may be designated as read and write, read only, write only, or no access. This designation is a requirement on a CIM schema implementation to constrain the ability to access the property's values as specified, and does not imply authorization to access those values.

5.6.3.3 Property override

A property may *override a property* with the same name that is defined in a supertype of the containing type. Such a property in the subtype is called the *overriding* property, and the designated property is called the *overridden* property.

Qualifiers of the overridden property are propagated to the overriding property as described in 5.6.12.

The overriding and the overridden properties shall be consistent, as follows:

- The type of a structure, (including class and association), typed property shall be the same as, or a subtype of the overridden property.
- The type of an enumeration typed property shall be the same as, or a supertype of the overridden property.
- The type of a primitive typed property shall be the same as the overridden property.
- The overridden and overriding property shall be both array or both scalar.

An overridden property is not exposed. An overriding property is exposed and inherits the qualifiers of the overridden property as described in 5.6.12.

5.6.3.4 Reference property

A reference property is a property that has a type that is declared as a reference to a named class, and has values that reference instances of that class (this includes instances of its subclasses).

A reference property is handled differently depending on whether it belongs to an association or not.

A reference property declared in a structure or non-association class shall be either a scalar or an array.

A reference property declared in an association shall be a scalar; for more details see 5.6.8.

5.6.4 Method

A method specifies a behavior of a class. It shall have a unique name within the methods of its defining class, including any inherited method. A method may have zero or more qualifiers (see 5.6.12), some of which apply specifically to the method return, while others apply to the method as a whole.

Method invocations can cause changes in property values of the defining class instance and might also affect changes in the modeled system and as a result in the existence or values of other instances.

The following ABNF defines the syntax for method names.

```
methodName = IDENTIFIER
```

A method may have at most one method return that may be a scalar or array. If none, the method is said to be "void". The method return defines the type of the return value passed out of a method.

A method may have zero or more parameters (see 5.6.5).

A method may be designated as static.

A non-static method can be invoked on an instance of the class in which it is defined or its subclasses.

A static method can be invoked on class in which it is defined, on a subclass of that class or on an instance of that class or its subclasses. When invoked on an instance, a CIM schema implementation of a static method shall not depend on the state of that instance.

5.6.4.1 Method override

A method may *override a* method with the same name that is defined in a superclass of the containing class. Such a method in the subclass is called the *overriding* method, and the designated method is called the *overridden* method.

Qualifiers of the overridden method (including its parameters) are propagated to the overriding method as described in 5.6.12.

The return values of overriding and the overridden methods shall be consistent, as follows:

- The return type of an overriding method that has a return type of a structure (including a class or association) shall be the same as or a subtype of the return type of the overridden method.
- The return type of an overriding method that has a return type of an enumeration shall be the same as or a supertype of the return type of the overridden method.
- The return type of an overriding method that has a return type of a primitive type shall be the same as the return type of the overridden method.
- The overridden and overriding method return shall be both array or both scalar.

The parameter having the same name in both an overriding and overridden method shall be consistent, as follows:

- An input parameter of an overriding method that has a type of
  - a structure (including a class or association) shall be the same as, or a supertype of, the type of the overridden parameter
  - an enumeration shall be the same as, or a subtype of, the type of the overridden parameter
  - a primitive type shall be the same as the type of the overridden parameter
- An output parameter of an overriding method that has a type of
  - a structure (including a class or association) shall be the same as, or a subtype of, the type of the overridden parameter
  - an enumeration shall be the same as, or a supertype of, the type of the overridden parameter
  - a primitive type shall be the same as the type of the overridden parameter
- A parameter of an overriding method that is both input and output shall be the same as the type of the overridden parameter.
- The overridden and overriding parameter shall be both array or both scalar.

An overridden method is not exposed by the overriding class or association. An overriding method is exposed and inherits the qualifiers of the overridden method as described in 5.6.12.

5.6.5 Parameter

A parameter is a named and typed specification of an argument passed into or out of an invocation of a method. Each parameter has a name that is unique within the method and zero or more qualifiers (see 5.6.12). The following ABNF defines the syntax for parameter names.

```
parameterName = IDENTIFIER
```

A parameter may be a scalar or an array.

A parameter has a direction (input, output, or both).

An input parameter that specifies a default value is referred to as optional. Optional parameters may be omitted on a method invocation. If omitted, a CIM schema implementation shall assume the default.

5.6.6 Structure

A structure is a type that models a complex value. A structure has zero or more properties (see 5.6.3) and zero or more qualifiers (see 5.6.12).

A structure shall not have methods.

A structure may be defined at the schema level with a schema-unique name or within a structure, class, or association with a structure-unique name (see 5.7.2). The name of a structure is used as its type name. The following ABNF defines the syntax for local and schema level structure names.

```
localStructureName = IDENTIFIER

structureName = schemaName "_" IDENTIFIER
```

A structure may define structures and enumerations (see 5.6.1). Such structure and enumeration definitions are called local. Local structures and enumerations can be used as the types of elements in their defining structure or its subtypes, but they cannot be used outside of their defining structure and its subtypes.

A structure may directly inherit from one other structure. A structure (not a class) shall not inherit from a class.

In an inheritance relationship between structures, the more general structure is called the *supertype*, and the more specialized structure is called the *subtype*.

The set of properties defined and inherited is called the set of properties *exposed* by the structure.

If a structure has a supertype, all properties exposed by the supertype are inherited by the structure. The subtype then has both the properties it defines and the inherited properties. See 5.6.3.3 for a discussion about the overridden properties.

A structure may be abstract. Abstract structures cannot be used as types of elements.

5.6.7 **Class**

A class models an aspect of a managed resource. A class is a type that has zero or more properties, methods, and qualifiers and may define local structures and enumerations (see 5.6.1). Unless defined differently, all of the rules for structures (see 5.6.6) apply to classes. The methods of a class represent exposed behaviors of the managed resource it models, and its properties represent the exposed state or status of that resource.

A class shall be defined at the schema level. Within that schema, the class name shall be unique (see 5.7.2) and is used as its type name. The following ABNF defines the syntax for class names.

```
className = schemaName "_" IDENTIFIER
```

A class may inherit from either one structure or from one class. In an inheritance relationship between classes, the more general class is called the *superclass*, and the more specialized type is called the *subclass*.

A class (not an association) shall not inherit from an association.

The set of methods defined and inherited is called the set of methods *exposed* by the subclass.

If a class has a superclass, all methods exposed by the superclass are inherited by the class. The subclass then has both the elements it defines and the inherited elements. See clause 5.6.4.1 for a discussion of method overriding.

A class may be abstract. Abstract classes cannot have instances and cannot be used as a type of an element. Concrete classes shall expose one or more key properties; abstract classes may expose one or more key properties.

A realization of a concrete class is a separately addressable instance.

The class name and the name value pairs of all key properties in an instance shall uniquely identify that instance in the scope in which it is instantiated.

The values of key properties are determined once at instance creation time and shall not be modified afterwards. For a comparison of instance values, see ANNEX C.

The value of a property in an instance of a class shall be consistent with the declared type of the property. If the property is required (see 7.25), then its value shall be non-Null; otherwise, it may be Null.

5.6.8 **Association**

An association is a type that models the relationship between two or more managed resources. An association instance represents a relationship between instances of the related classes. The related classes are specified by the reference properties of the association.

The semantics of an association are different from that of a class having one or more properties of type reference. In an association, all references are endpoints of the associations. In a class, each reference is an independent pointer to an instance.

In an association each reference property shall be a scalar and all reference properties shall not be Null.

An association has zero or more properties, methods, and qualifiers and may define local structures and enumerations (see 5.6.1). Unless defined differently, all of the rules for classes (see 5.6.7) apply to associations. The name of an association is used as its type name.

References, as with all properties of an association, are members of the association.

The reference properties may also be keys of an association. In associations, where the set of references are all keys and no other properties are keys, at most one instance is possible between a unique set of referenced instances. Otherwise it is possible to have multiple association instances between the same set of instances.

The values of reference properties are determined once at instance creation time and shall not be modified afterwards.

The multiplicity in the relationship between associated instances is specified on the reference properties of the association, such that the multiplicity specified on a particular reference property is the range of the number of instances that can be associated to a unique combination of instances referenced by the other reference properties.

EXAMPLE 1: Given a binary association with reference properties a and b. If b has multiplicity 1..2, then for a set of association instances: for each instance referenced by a; the set of instances referenced by b must include at least one instance and no more than 2.

EXAMPLE 2: Given a ternary association with reference properties a, b, and c. If b has multiplicity [1..2], then for a set of association instances: for each unique pair of instances referenced by a and c; b must reference at least one instance and no more than 2.

NOTE 1 For all association instances, at least two reference properties must not be Null.

NOTE 2 In an instance of a ternary or above association, the value of a reference property may be Null if its multiplicity lower bound is zero (0) and it is not qualified as Required (see 7.25) and at least two other reference properties have values that are not Null.

The association name of an association defined at the schema level, shall be unique (see 5.7.2) and is used as its type name. The following ABNF defines the syntax for association names.

```
associationName = schemaName "_" IDENTIFIER
```

An association may inherit from one other association. In an inheritance relationship between associations, the more general association is called the *superclass*, and the more specialized type is called the *subclass*.

A subclass of an association shall not change the number of reference properties.

In the case when the relationship is binary (i.e., between only two classes), the reference properties of an association may additionally indicate that instances of one (aggregated) class are aggregated into instances of the other (aggregating) class. There are two types of aggregation.

- Shared aggregation indicates that the aggregated instances may be aggregated into more than one aggregating instances. In this case, the referenced instance generally has a lifecycle that is independent of referencing instances.
- Composite aggregation indicates that referenced instances are part of at most one referencing instance. Unless removed before deletion, referenced instances are typically deleted with the referencing instance. However, that policy is left to be specified as semantics of the modeled elements.

5.6.9 Reference type

A reference type models a reference to an instance of a specified class, including to instances of subclasses of the specified class. The name of a ReferenceType is used as its type name.

For two classes, C1 and C2, and corresponding reference types defined on those classes, R1 and R2: R2 is a subtype of R1 if C2 is a subclass of C1.

The referenced class may be abstract; however, all values shall refer to instances of concrete (non-abstract) classes. The classes of these instances may be subclasses of the referenced class. As a result, all reference types are concrete.

5.6.10 Instance value

An instance value represents the specification of an instance of a class or association.

For a comparison of the specification of instances, see ANNEX C.

5.6.11 Structure value

A structure value is a model element that specifies the existence of a value for a structure.

For comparison of structure values, see ANNEX C.

5.6.12 Qualifier types and qualifiers

Qualifier types and qualifiers provide a means to add metadata to schema elements.

Some qualifier types and qualifiers affect the schema element's behavior, or provide information about the schema element.

A qualifier type is a definition of a qualifier in the context of a schema. Defining a qualifier type in a schema effectively adds a metadata attribute to every element in its scope with a value that is the default value defined by the qualifier type. A qualifier type specifies a name, type, default value, propagation policy, and scope.

Qualifier scope is a list of schema element types. A qualifier shall be applied only to schema elements listed in the scope of its qualifier type.

When adding a qualifier type to a schema, its default value should not change the existing behavior of the schema elements in its scope.

A qualifier type shall be defined at the schema level. Within that schema, the qualifier type name shall be unique (see 5.7.2). The following ABNF defines the syntax for qualifier type names.

```
qualifierTypeName = [schemaName "_"] IDENTIFIER
```

Except for qualifier types defined by this specification, the use of the optional schemaName is strongly encouraged. The use of the schemaName assures that extension schema defined qualifiers will not conflict with qualifiers defined by this specification or with those defined in other extension schemas.

A qualifier provides a means to modify the value of the metadata attribute defined by the default value of the qualifier type.

The propagation policy controls how the value of an applied qualifier is propagated to affected elements in subclasses. There are three propagation policies.

- restricted

- disableOverride
- enableOverride

The "restricted" propagation policy specifies that the value of an applied qualifier does not propagate to elements in the propagation graph as defined in Table 5. Instead, and unless qualified directly, the behavior of elements lower in the propagation graph is as if the default value of the qualifier type was applied. A "restricted" qualifier may be specified anywhere in an element's propagation graph.

The "disableOverride" propagation policy specifies that the element at the top of the propagation graph has either the default value or a specified value for this qualifier. Each element lower in the propagation graph has the same value and that value cannot be changed. A "disableOverride" qualifier may be re-specified lower in the propagation graph, but shall not change the value.

The "enableOverride" propagation policy specifies that the qualifier may be specified on any element in a propagation graph. For elements higher than the first application of the qualifier in the propagation graph, the qualifier has the default value of its qualifier type.

NOTE 1 In the propagation graph higher means towards supertypes and lower means towards subtypes.

NOTE 2 Propagation is towards elements lower in the propagation graph.

Table 5 – Propagation graph for qualifier values

Qualified Element	Elements in the Propagation Graph
Association	Sub associations
Class	Sub classes
Enumeration	Sub enumerations
Enumeration value	Like named enumeration values of sub enumerations
Method	Overriding methods of sub classes (including associations)
Parameters	Like named parameters of overriding methods of sub classes (including associations)
Property	Overriding properties of sub structures (including classes and associations)
Qualifier type	Not applicable
Reference	Overriding references of sub structures (including classes and associations)
Structure	Sub structures (including associations and classes)

Qualifier types are defined in clause 7.

5.7 Naming of model elements in a schema

5.7.1 Matching

Element names are matched case insensitively.

CIM Metamodel implementations shall preserve the case of element names.

5.7.2 Uniqueness

Model element names are defined in the context of an element that serves as a naming context.

Each schema level element (structure, class, association, enumeration, qualifier type, instance value and structure value) name shall be unique within the set of schema level elements exposed by its schema.

Each locally defined type (structure or enumeration) name shall be unique within the set of local defined type names exposed by its structure, class or association.

Each enumeration value name shall be unique within the set of enumeration value names exposed by its enumeration.

Each property name shall be unique within the set of property names exposed by its structure, class or association.

Each method name shall be unique within the set of method names exposed by its class or association.

Each parameter name shall be unique within the set of parameter names exposed by its method.

5.8 Schema backwards compatibility rules

This clause defines rules for modifications that assure backwards compatibility for clients.

NOTE Additional rules for qualifiers are listed in clause 7.

Table 6 describes modifications that are backwards compatible for clients.

NOTE The table is organized into simple cases that can be combined.

Table 7 describes schema modifications that are not backwards compatible for clients.

Table 6 – Backwards compatible schema modifications

ID	Modification
C1	Adding a class to the schema. The new class may inherit from an existing class or structure.
C2	Adding a structure to the schema or as a local definition to a structure, class, or association. The new structure may inherit from an existing structure.
C3	Adding an enumeration to the schema or as a local definition to a structure, class, or association. The new enumeration may inherit from an existing enumeration.
C4	Adding an association to the schema. The new association may inherit from an existing association.
C5	Inserting a class into an inheritance hierarchy of existing classes (see also C6, C7, C9, and C10).
C6	Adding a property to an existing class that is not overriding a property. The property may have a non-Null default value.
C7	Adding a property to an existing structure, class or association that is overriding a property.
C8	The overriding property specifies a type or qualifier that is compatible with the overridden property, see Table 7
C9	The overriding property specifies a default value that is different from the default value specified by the overridden property.
C10	Moving an existing property from a structure, class or association to one of its super classes.
C11	Adding a method to an existing class or association that is not overriding a method.
C12	Adding a method to an existing class or association that is overriding a method.
C13	The overriding method specifies changes to the type or qualifiers applied to the method or its parameters that are compatible with the overridden method or its parameters, see Table 7
C14	Moving a method from a class or association to one of its super classes.
C15	Adding an input parameter to a method with a default value.
C16	Adding an output parameter to a method.
C17	Changing the effective value of a qualifier type on an existing schema element depends on definition of qualifier types and on the allowed qualifier type modifications listed in Table 7.

ID	Modification
C18	Changing the complex type (i.e., structure, class, or association) of an output parameter, method return, or property to a subtype of that complex type.
C19	Changing the enumeration type of an output parameter, method return, or property to a supertype of that enumeration type.
C20	Changing the complex type (i.e., structure, class, or association) of an input parameter to a supertype of that complex type.
C21	Changing the enumeration type of an input parameter to a subtype of that enumeration type.
C22	Adding an enumeration value to an enumeration.
C23	Restricting the allowable range of values (including disallowing Null if previously allowed), for output parameters and method return or readable properties.

Table 7 – Schema modifications that are not backwards compatible

ID	Modification
I1	Removing a structure, class, association or enumeration from the schema.
I2	Changing the supertype of type such that it is no longer a subtype of the original supertype.
I3	Changing a concrete type to be abstract.
I4	Changing key property to be a non-key property or vice-versa.
I5	Removing a local structure, local enumeration, property or method from an existing type, without adding it to one of its super types.
I6	Changing the complex type (i.e., structure, class, or association) of an output parameter, method return, or property to a supertype of that complex type.
I7	Changing the enumeration type of an output parameter, method return, or property to a subtype of that enumeration type.
I8	Changing the complex type (i.e., structure, class, or association) of an input parameter to a subtype of that complex type.
I9	Changing the enumeration type of an input parameter to a supertype of that enumeration type.
I10	Removing an enumeration value from an enumeration.
I11	Changing the value of an enumeration value in an enumeration.
I12	Removing an input or output parameter.
I13	Changing the direction of a parameter (including, for example, changes from in to inout).

ID	Modification
I14	Adding an input parameter to an existing method that has no default.
I15	Removing a parameter from an existing method.
I16	Changing the primitive type of an existing method parameter, method (i.e., its return value), or ordinary property.
I17	Changing a reference property, parameter or method return to refer to a different class.
I18	Changing a meta type of a type (i.e., between structure and class or class and association).
I19	Reducing or increasing the arity of an association (i.e., increasing or decreasing the number of references exposed by the association).
I20	Increasing the allowable range of values (including allowing Null if previously disallowed), for output parameters and method return or readable properties.
I21	Restricting the allowable range of values for input parameters or writeable properties (including disallowance of Null if it had been allowed).
I22	Removing a qualifier type declaration.
I23	Changing the datatype or multiplicity of an existing qualifier type declaration.
I24	Removing an element type from the scope of an existing qualifier type declaration.
I25	Changing the propagation policy of an existing qualifier type declaration.
I26	Adding a qualifier type declaration if the default value implies a change to affected schema elements.
I27	Adding an element type to the scope of an existing qualifier type declaration if the default value implies a change to affected schema elements.

6 CIM metamodel

This clause normatively defines the semantics, attributes, and behaviors of the elements that comprise the CIM Metamodel. CIM Metamodel is specified as a UML user model (see the [Unified Modeling Language: Superstructure]() specification). The principal elements of the CIM Metamodel are normatively shown in Figure 1.

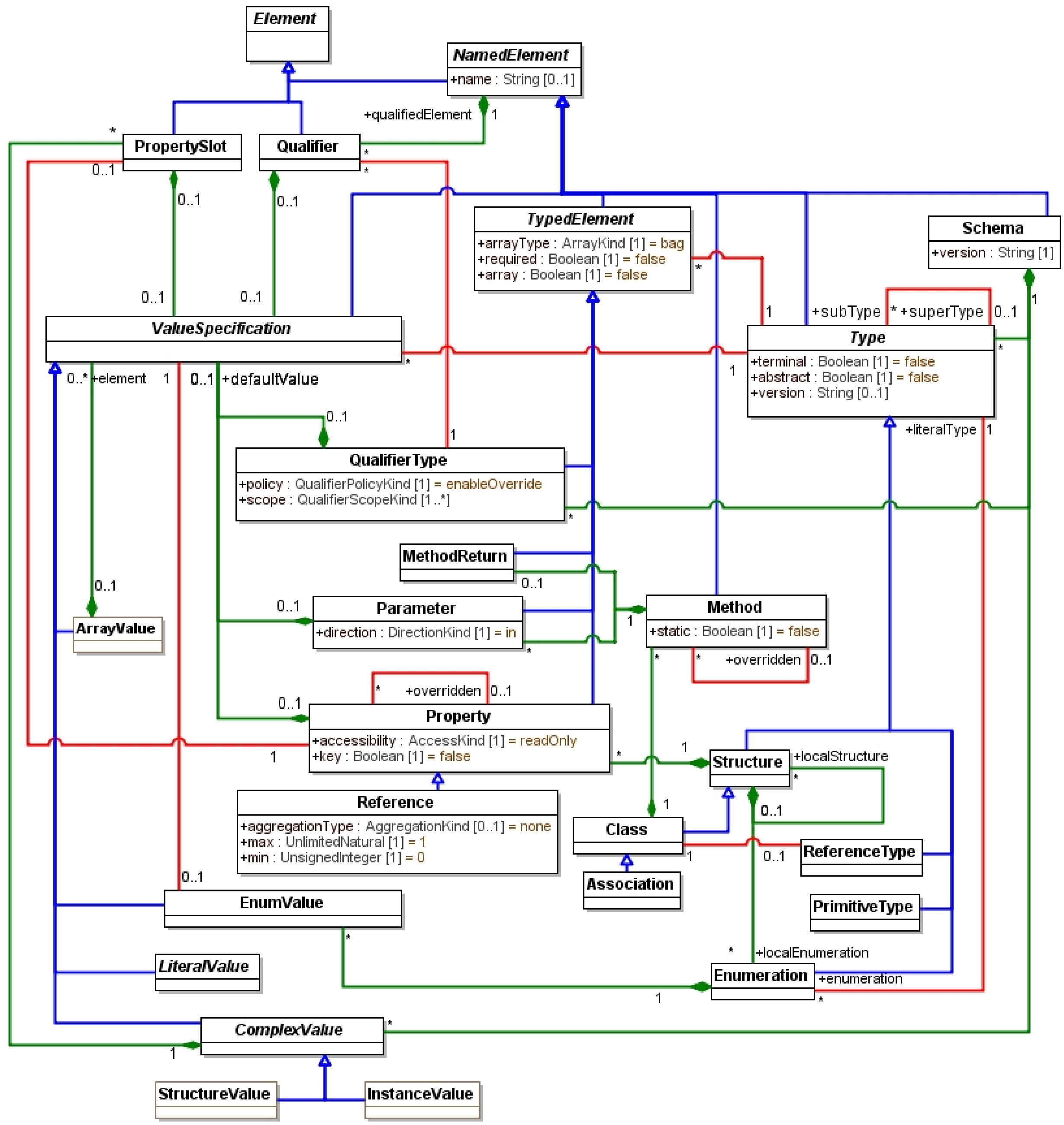

Figure 1 – Overview of CIM Metamodel

6.1 Introduction

The CIM Metamodel is the basis on which CIM schemas are defined.

This clause specifies concepts used across the specification of the metamodel and assumes some familiarity with UML notation and with basic object-oriented concepts.

A subset of the OMG Object Constraint Language (OCL) is used to precisely specify constraints on the metamodel. That subset is defined in clause 8.

CIM Metamodel implementations shall support the semantics and behaviors specified in this document. However, there is no requirement for CIM Metamodel implementations to implement the metaelements described here.

The metaelements shown in Figure 1 are just one way to represent the semantics of the CIM Metamodel. Other choices could have been made without changing the semantics; for example, by moving associations between metaelements up or down in the inheritance hierarchy, or by adding redundant associations, or by shaping the attributes differently. However, one way of shaping the metaelements had to be picked to normatively express the semantics of the CIM Metamodel. The key requirement on any representation is that it expresses all of the requirements and constraints of the CIM Metamodel.

In this document, when it is important to be clear that a CIM Metamodel metaelement is being referred to, the name of the metaelement will be prefixed by "CIMM::". For instance, CIMM::Association refers to the CIM Metamodel element named Association.

6.2 Notation

The following clauses describe additional rules on the usage of UML for specification of the CIM Metamodel.

6.2.1 Attributes

Descriptions of attributes throughout clause 6 use the attrFormat ABNF rule (whitespace allowed):

```
attrEnum = IDENTIFIER

attrDefault = ( Null / "true" /  "false"  / "0" / "1" / attrEnum)

attrMultiplicity = multiplicity

attrType = IDENTIFIER

attrName = IDENTIFIER

attrFormat = attrName ":" attrType [ "[" attrMultiplicity "]" ]
             [ "=" attrDefault ]
```

NOTE Multiplicity specifies the valid cardinalities for values of the attribute. A lower bound of zero indicates that the attribute may be Null, (i.e., no value). If the lower bound is specified as zero and a default value is specified, then the attribute must be explicitly set to be Null.

6.2.2 Associations

A relationship between metaelements is modeled as a UML association. In this metamodel, association ends are owned by the associated elements and the association has no additional properties. As a consequence, association ends are listed with their owning metaelements and associations are not listed as separate metaelements.

Descriptions of association ends within the metamodel use the associationEndFormat ABNF rule (whitespace allowed):

```
associationEndFormat = otherRole ":" otherElement
                            "[" other-cardinality "]"
otherRole = IDENTIFIER
otherElement = IDENTIFIER
otherMultiplicity = multiplicity
```

6.2.3 Constraints

Constraints on CIM Metamodel are defined on the metaelements that define the metamodel. CIM Metamodel implementations shall enforce the specified constraints.

These constraints fall into two categories:

- OCL constraints – Constraints defined by using a subset of the Object Constraint Language (OCL) as defined in clause 8. This is the main category of constraints unless otherwise specified:
  - The OCL context (i.e., self) for resolving names is the constrained metamodel element.
  - Unless needed for clarity, "self" is not explicitly stated and is assumed to prefix all names used in an OCL constraint according to the following ABNF:

    ```
    name = [ "self." ] IDENTIFIER  * ("." IDENTIFIER)
    ```

  - All constraints are invariant and the `context` and `inv` keywords are implied and not stated.
- Other constraints – Constraints defined by using normative text. This category only exists for constraints for which it was not possible to define an according OCL statement.

NOTE OCL is used as a specification language in this document. CIM Metamodel implementations may use other OCL statements or constraint languages other than OCL as long as they produce an equivalent result.

6.3 Types used within the metamodel

The following types are used within the metamodel.

6.3.1 AccessKind

AccessKind is an enumeration for specifying a property's ability to read and write its value.

Table 8 – AccessKind

Enumeration value	Description
noAccess	No access
readOnly	Read only access
readWrite	Read and write access
writeOnly	Write only access

6.3.2 AggregationKind

AggregationKind specifies whether the relationship between two or more schema elements is: not an aggregation; is a shared aggregation; or is a composite aggregation (see 5.6.8). AggregationKind is specified on one end of an association.

Table 9 – AggregationKind

Enumeration value	Description
None	The relationship is not an aggregation.
Shared	The relationship is a shared aggregation.
Composite	The relationship is a composite aggregation.

6.3.3 ArrayKind

ArrayKind (see Table 3) is an enumeration for specifying the characteristics of the elements of an array.

6.3.4 Boolean

An element with a true or false value.

6.3.5 DirectionKind

DirectionKind is an enumeration used to specify direction of parameters.

Table 10 – DirectionKind

Enumeration value	Description
In	The parameter direction is input.
inout	The parameter direction is both input and output.
out	The parameter direction is output.

6.3.6 PropagationPolicyKind

PropagationPolicyKind is an enumeration for defining QualifierType value change policies (see 5.6.12).

Table 11 – PropagationPolicyKind

Enumeration value	Description
disableOverride	Indicates a qualifier type’s propagation policy is disableOverride
enableOverride	Indicates a qualifier type’s propagation policy is enableOverride
restricted	Indicates a qualifier type’s propagation policy is restricted

6.3.7 QualifierScopeKind

QualifierScopeKind is an enumeration that defines the metaelements that may be in a QualifierType’s scope (see 5.6.12).

Table 12 – QualifierScopeKind

Enumeration value	Description
association	Qualifiers may be applied to associations.
class	Qualifiers may be applied to classes.
enumeration	Qualifiers may be applied to enumerations.
enumValue	Qualifiers may be applied to enumeration value specifications.
method	Qualifiers may be applied to methods, including method returns.
parameter	Qualifiers may be applied to parameters.
property	Qualifiers may be applied to properties.
qualifierType	Qualifiers may be applied to qualifier types.
reference	Qualifiers may be applied to reference properties, including in both associations and classes.
structure	Qualifiers may be applied to structures.
any	Qualifiers may be applied to all other enumerated elements.

6.3.8 **String**

A string is a sequence of characters in some suitable character set that is used to display information about the model (see 5.5.3).

6.3.9 **UnlimitedNatural**

An unlimitedNatural is an element in the set of non-negative integers (0, 1, 2…) and unlimited. The value of unlimited is shown using an asterisk (‘*’).

6.3.10 **UnSignedInteger**

An unsignedInteger is an element in the set of non-negative integers (0, 1, 2…).

6.4 Metaelements

6.4.1 **CIMM::ArrayValue**

An ArrayValue is a metaelement that represents a value consisting of a sequence of zero or more element ValueSpecifications of same type.

Generalization

CIMM::ValueSpecification (see 6.4.26)

Attributes

No additional attributes

Associations

- The ValueSpecifications that are the values of elements of the array

```
element: ValueSpecification [0..*]
```

Constraints

Constraint 6.4.1-1: An ArrayValue shall have array type

```
type.array
```

Constraint 6.4.1-2: The elements of an ArrayValue shall have scalar type

```
element->forAll(v | not v.type.array)
```

6.4.2 **CIMM::Association**

The Association metaelement represents an association (see 5.6.8).

Generalization

CIMM::Class (see 6.4.3)

Attributes

No additional attributes

Associations

No additional associations

Constraints

Constraint 6.4.2-1: An association shall only inherit from an association

```
superType->NotEmpty() implies superType.oclIsKindOf(Association)
```

Constraint 6.4.2-2: A specialized association shall have the same number of reference properties as its superclass

```
superType->select( g | g.oclIsKindOf(Association))->notEmpty() implies
   superType->property->select( pp | pp.oclIsKindOf(Reference))->size() =
            property->select( pc | pc.oclIsKindOf(Reference))->size()
```

Constraint 6.4.2-3: An association class cannot reference itself.

```
property->select( p | p.oclIsKindOf(Reference))->type->forAll(t | t.class-
>excludes(self)) and
property->select( p | p.oclIsKindOf(Reference))->type->
   forAll(t | t.class ->collect(et|et.allSuperTypes()->excludes(self)))
```

Constraint 6.4.2-4: An association class shall have two or more reference properties

```
property->select( p | p.oclIsKindOf(Reference)) ->size() >= 2
```

Constraint 6.4.2-5: The reference properties of an association class shall not be Null

```
property->select( p | p.oclIsKindOf(Reference) and not p.oclIsUndefined())
```

6.4.3 **CIMM::Class**

The Class metaelement models a class (see 5.6.7).

Generalization

CIMM::Structure (see 6.4.22)

Attributes

No additional attributes

Associations

- ReferenceType that refers to this class

```
referenceType : ReferenceType [0..1]
```

- Methods owned by this class

```
method : Method[0..*]
```

Constraints

Constraint 6.4.3-1: All methods of a class shall have unique, case insensitive names.

```
self.exposedMethods()->
   ; iterate through all exposed methods and check that names are distinct.
   forAll( memb | self.exposedmethods->excluding(memb)->
         forAll( other |  memb.name.toUpperCase() <> other.name.toUpperCase() )
)
```

Constraint 6.4.3-2: If a class is not abstract, then at least one property shall be designated as a Key

```
not abstract implies select(exposedProperties()->key).size() >= 1
```

Constraint 6.4.3-3: A class shall not inherit from an association.

```
superType->notEmpty() and not self.oclIsKindOf(Association)
   implies not superType->forAll(g | g.oclIsKindOf(Association))
```

Operations

- The exposedMethods operation includes all exposed methods in the inheritance graph.

```
Class::exposedMethods() : Set(Method);
exposedMethods = method->union(allSuperTypes()->method)
```

6.4.4 CIMM::ComplexValue

A ComplexValue is a metaelement that is the abstract base class for the metaelements StructureValue and InstanceValue.

Generalization

CIMM::ValueSpecification (see 6.4.26)

Attributes

No additional attributes

Associations

- A ComplexValue is defined in a Schema.

```
schema : Schema [1]
```

- Each propertySlot gives the value or values for each represented property of the defining class or structure.

```
propertySlot: PropertySlot [0..*]
```

Constraints

No additional constraints

6.4.5 CIMM::Element

Element is an abstract metaelement common to all other metaelements.

Generalization

None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

6.4.6 CIMM::Enumeration

An Enumeration metaelement models an enumeration (see 5.6.1).

Generalization

CIMM::Type (see 6.4.24)

Attributes

No additional attributes

Associations

- An Enumeration has a literal type.

```
literalType: Type[1]
```

- A local Enumeration belongs to a Structure.

```
structure : Structure[0..1]
```

- An Enumeration is the scoping element for enumeration values.

```
enumValue : EnumValue[0..*]
```

Constraints

Constraint 6.4.6-1: All enumeration values of an enumeration have unique, case insensitive names.

```
Let el = self.exposedValues() in
el->forAll( memb |
   el->excluding(memb)->
      forAll( other |  memb.name.toUpperCase() <> other.name.toUpperCase() )  )
```

Constraint 6.4.6-2: The literal type of an enumeration shall not change through specialization

```
superType->notEmpty() implies literalType=superType.literalType
```

Constraint 6.4.6-3: The literal type of an enumeration shall be a kind of integer or string

NOTE integer includes signed and unsigned integers

```
literalType.oclIsKindOf(integer) OR literalType.oclIsKindOf(string))
```

Constraint 6.4.6-4: Each enumeration value shall have a unique value of the enumeration's type

```
Let elv = self.exposedValues()->valueSpecification in
   If self.literalType.oclIsKindOf(string) then
      elv->forAll(v | v->size()=1 and v.oclIsKindOf(StringValue)) and
      elv->forAll(memb | elv->excluding(memb)->
              forAll(other | memb.oclAsKindOf(StringValue).value<>
                      other.oclAsKindOf(StringValue).value)))
   else – integer
      elv->forAll(v | v->size()=1 and v.oclIsKindOf(IntegerValue)) and
      elv->forAll(memb | elv->excluding(memb)->
             forAll(other | memb.oclAsKindOf(IntegerValue).value<>
                     other.oclAsKindOf(IntegerValue).value)))
   endif
```

Constraint 6.4.6-5: The super type of an enumeration shall only be another enumeration

```
superType->notEmpty() implies superType.oclIsKindOf(Enumeration)
```

Constraint 6.4.6-6: An enumeration with zero exposed enumeration values shall be abstract

```
self.exposedValues()->size()=0 implies abstract
```

Operations

- The exposedValues operation excludes overridden enumeration values.

```
Enumeration::exposedValues() : Set(EnumValue);
If superType.isEmpty() then
   exposedValues = enumValue
else
   exposedValues = enumValue->
      union(superType->exposedValues()->excluding(enumValue))
```

6.4.7 CIMM::EnumValue

The enumeration value metaelement models a value of an enumeration (see 5.6.2).

Generalization

CIMM::ValueSpecification (see 6.4.26)

Attributes

No additional attributes

Associations

- Enumeration value is defined in an Enumeration.

```
enumeration : Enumeration [1]
```

- An enumeration value has a value.

  NOTE The default for a string enumeration value is its name and it is resolved at definition time.

```
valueSpecification : ValueSpecification [1]
```

Constraints

Constraint 6.4.7-1: Value of string enumeration is a StringValue; Null not allowed.

```
enumeration.oclIsKindOf(string) implies
valueSpecification.oclIsKindOf(StringValue)
```

Constraint 6.4.7-2: Value of an integer enumeration is a IntegerValue; Null not allowed.

```
enumeration.oclIsKindOf(integer) implies
valueSpecification.oclIsKindOf(IntegerValue)
```

6.4.8 CIMM::InstanceValue

An InstanceValue is a metaelement that models the specification of an instance (see 5.6.10).

When used as the value or default value of a typed element an InstanceValue shall not be abstract. The type of the InstanceValue shall be the same as, or a subclass of, that element's type.

Generalization

CIMM::ComplexValue (see 6.4.4)

Attributes

No additional attributes

Associations

No additional associations

Constraints

- Constraint 6.4.8-1: An InstanceValue has the type of a class or association

```
type.oclIsKindOf(Class)
```

6.4.9 CIMM::LiteralValue

A LiteralValue is an abstract metaelement that models the specification of a value for a typed element in the range of a particular primitive type or in the case of NullValue represents that the typed element is Null, (see 5.5.4).

LiteralValue has specialized metaelements for each primitive type. Each of the subtypes, except for NullValue, has a value attribute that are used to represent a value of a primitive type.

The concrete subclasses of LiteralValue are shown in Table 13.

Table 13 – Specializations of LiteralValue

Subclasses	Interpretation
BooleanValue	A non-Null value of type boolean as defined in Table 4
DateTimeValue	A non-Null value of type datetime as defined in 5.5.1
IntegerValue	A non-Null value of one of the concrete subtypes of abstract type integer as defined in Table 4
NullValue	Represents the state of Null as defined in 5.5.4
OctetStringValue	A non-Null value of type octetstring defined as in 5.5.2
RealValue	A non-Null value of one of the concrete subtypes of abstract type real defined in Table 4
ReferenceValue	A non-Null value of type reference defined in 5.6.9
StringValue	A non-Null value of type string defined in 5.5.3

Generalization

CIMM::ValueSpecification (see 6.4.26)

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

6.4.10 CIMM::Method

The Method metaelement models methods in classes and associations (see 5.6.4)

Generalization

CIMM::NamedElement (see 6.4.12)

Attributes

- static indicates if the method is static. The value is determined by the Static qualifier.

```
static : boolean [1]
```

Associations

- Class that owns this method

```
class: Class [1]
```

- A method return of this method

```
methodReturn : MethodReturn [0..1]
```

- Parameters of this method

```
parameter: Parameter [0..*]
```

- Methods that override this method

```
method: Method [0..*]
```

- A method that is overridden by this method

```
overridden : Method [0..1]
```

Constraints

Constraint 6.4.10-1: All parameters of the method have unique, case insensitive names.

```
parameter->forAll( memb |  parameter->excluding(memb)->
      forAll( other |  memb.name.toUpperCase() <> other.name.toUpperCase() )  )
```

Constraint 6.4.10-2: A method shall only override a method of the same name.

```
overridden->notEmpty() implies
   overridden.oclIsKindOf(Method)and name->toUpper() = overridden.name-
>toUpper()
```

Constraint 6.4.10-3: A method return shall not be removed by an overriding method (changed to void).

```
overridden->notEmpty() and methodReturn.isEmpty() implies
    overridden.methodReturn.isEmpty()
```

Constraint 6.4.10-4: An overriding method shall have at least the same method return as the method it overrides.

```
overridden.notEmpty() and methodReturn.notEmpty() implies
      overridden.methodReturn.notEmpty() and
      methodReturn.type->oclIsKindOf(overridden.parameter.type) and
      methodReturn.array = overridden.methodReturn.array
```

Constraint 6.4.10-5: An overriding method shall have at least the same parameters as the method it overrides.

Additional out Parameters are allowed and additional in or inout Parameters are allowed if a default value is specified.

```
overridden.notEmpty() implies
   parameter->size() >= overridden.parameter->size() and
   let oldParm = overridden.parameter in
   let newParm = parameter->excluding(oldParm) in
   ( oldParm->exists (op | parameter->exists(np |
         np->toUpper() = op.name->toUpper() and
         np.type.array = op.type.array and
         np.type.direction = op.type.direction and
         (
         -- A input parameter of an overriding method that has a type of a
         -- structure (including a class or association) shall be the same
         -- as or a supertype of the type of the overridden parameter
         (np.type.oclIsKindOf(Structure) and np.direction = DirectionKind.in
          and op.type.oclIsKindOf(np.type))
         or
         -- A input parameter of an overriding method that has a type of an
```

```
            -- enumeration shall be the same as or a subtype of the type of the
            -- overridden parameter
            (np.type.oclIsKindOf(Enumeration) and np.direction = DirectionKind.in
             and
            np.type.oclIsKindOf(op.type))
            or
            -- A output parameter of an overriding method that has a type of a
            -- structure (including a class or association) shall be the same as
            -- or a subtype of the type of the overridden parameter
            (np.type.oclIsKindOf(Structure) and np.direction = DirectionKind.out
             and
            np.type.oclIsKindOf(op.type))
            or
            -- A output parameter of an overriding method that has a type of an
            -- enumeration shall be the same as or a supertype of the type of the
            -- overridden parameter
            (np.type.oclIsKindOf(Enumeration) and
             np.direction = DirectionKind.out and
             op.type.oclIsKindOf(np.type))
            or
            -- A parameter of an overriding method that has a primitive type
            -- shall be the same as the type of the overridden parameter
            (np.type.oclIsKindOf(Primitive) and np.type = op.type)
             or
            -- A parameter of that has direction inout shall be the same type as
            -- the type of the overridden parameter
            (np.direction = DirectionKind.inout and np.type = op.type)
            )
        ) )
    )
    and
    (   -- new in/inout parameters shall have a specified default value.
        newParm->forAll(np |
            np.direction=DirectionKind.in or np.direction=DirectionKind.inout
            implies np.defaultValue.notEmpty() )
    )
```

Constraint 6.4.10-6: An overridden method must be inherited from a more general type.

```
if overridden->notEmpty() then
    -- collect all the supertypes
     class->collect(fc | allSuperTypes())->asSet()->collect(c | c.method))-
>includes(overridden))
```

6.4.11 CIMM::MethodReturn

A MethodReturn metaelement models method return (see 5.6.4).

Generalization

CIMM::TypedElement (see 6.4.25)

Attributes

No additional attributes

Associations

- The method that this method return belongs to

```
method: Method [1]
```

Constraints

No additional constraints

Operations

Determine the set of method returns overridden by this methodReturn.

```
MethodReturn:allOverridden (): Set(MethodReturn);
let o = method.overridden.methodReturn in
    allOverridden =  o->union(o->collect(r | r.allOverridden() ))
```

6.4.12 CIMM::NamedElement

A NamedElement is an abstract metaelement that models elements that have a name.

Generalization

CIMM::Element (see 6.4.5)

Attributes

- A name of the realized element in the model

```
name : string [0..1]
```

Associations

- All applied qualifiers

```
qualifier : Qualifier [0..*]
```

Constraints

Constraint 6.4.12-1: Each qualifier applied to an element must have the element’s type in its scope.

```
qualifier.qualifierType->forAll(qt | qt.scope->includes(n | n->toUpper() =
oclIsKindOf(self)->toUpper())
```

6.4.13 CIMM::Parameter

A Parameter is a metaelement that models a named parameter of a method (see 5.6.5).

Generalization

CIMM::TypedElement (see 6.4.25)

Attributes

- Indicates the direction of the parameter, that is whether it is being sent into or out of a method, or both. The value is determined by the In and Out qualifiers.

```
direction : DirectionKind [1]
```

Associations

- An optional specification of the default value

```
defaultValue: ValueSpecification [0..1]
```

- The method that this parameter belongs to

```
method: Method [1]
```

Constraints

No additional constraints

Operations

Determine the set of parameters overridden by this parameter.

```
Parameter:allOverridden(): Set(Parameter);
let o = method.overridden.parameter->select(p | p.name->toUpper()=self.name-
```

```
>toUpper()) in
    allOverridden =  o->union(o->collect(p | p.allOverridden()))
```

6.4.14 **CIMM::PrimitiveType**

PrimitiveType is a metaelement that models a primitive type (see 5.5).

Generalization

CIMM::Type (see 6.4.24)

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

6.4.15 **CIMM::Property**

A Property is a metaelement that models the properties of structures, classes and associations (see 5.6.3).

Generalization

CIMM::TypedElement (see 6.4.25)

Attributes

- Indicates that the property is a key property. The value is determined by Key qualifier.
  ```
  key : boolean [1]
  ```
- Indicates whether or not the values of the modeled property can be read or written. The value is determined by the Read and Write qualifiers.
  ```
  accessibility : CIMM::AccessKind [1]
  ```

Associations

- Default values
  ```
  defaultValue : ValueSpecification [0..1]
  ```
- Properties that override this property
  ```
  property : Property [0..*]
  ```
- A Property that is overridden by this property
  ```
  overridden : Property [0..1]
  ```
- The structure that owns this property
  ```
  structure : Structure [1]
  ```
- PropertySlot models the values of a property for an InstanceValue.
  ```
  propertySlot : PropertySlot [0..*]
  ```

Constraints

Constraint 6.4.15-1: An overridden property must be inherited from a more general type.

```
if overridden->notEmpty() then
   -- collect all the supertypes
```

```
    structure->collect(st: Structure | structure.allSuperTypes())->
       -- collect all of their properties and check that the overridden property
is in that collection.
             collect(p : Property | st.allProperties())->includes(overridden))
```

Constraint 6.4.15-2: An overriding property shall have the same name as the property it overrides.

```
overridden->notEmpty() implies    name->toUpper() = overridden.name->toUpper()
```

Constraint 6.4.15-3: An overriding property shall specify a type that is consistent with the property it overrides (see 5.6.3.3).

```
overridden->notEmpty() implies
      type.oclIsKindOf(overridden.type)
```

Constraint 6.4.15-4: A key property shall not be modified, must belong to a class, must be of primitiveType, shall be a scalar value and shall not be Null.

```
key = true implies
   (accessibility = AccessKind::readOnly = true) and
   Structure.oclIsKindOf(Class) and
   type.oclIsKindOf(PrimitiveType) and   array = false and
   propertySlot->forAll(s | s->valueSpecification->size()=1 and
                  not s->valueSpecification.oclIsKindOf(NullValue))
```

Operations

- Determine the set of properties overridden by this property.

```
Property:allOverridden (): Set(Property);
allOverridden =   union(overridden->
      collect(p | p.allOverridden() and p.name->toUpper() = self.name-
>toUpper()))
```

6.4.16 CIMM::PropertySlot

A PropertySlot is a metaelement that models a collection of entries for a property in a complex value specification for the structure containing that property (see 5.6.10 and 5.6.11).

Generalization

CIMM::Element (see 6.4.5)

Attributes

No additional attributes

Associations

- The defining property for the values in the property slot of an InstanceValue

```
property : Property [1]
```

- The complexValue that owns this property slot

```
complexValue : ComplexValue [1]
```

- The value of the defining property

```
valueSpecification : ValueSpecification [0..1]
```

Constraints

Constraint 6.4.16-1: A scalar shall have at most one valueSpecification for its PropertySlot

```
property.type.array = false and valueSpecification.notEmpty() implies
valueSpecification.element.notEmpty()
```

Constraint 6.4.16-2: The values of a PropertySlot shall not be Null, unless the related property is allowed to be Null

```
valueSpecification->select (v | v.oclIsKindOf(NullValue))->notEmpty() implies
```

```
not property.required
```

Constraint 6.4.16-3: The values of a PropertySlot shall be consistent with the property type

```
let vs = valueSpecification->union(valueSpecification->element)->select(v | not
v.oclIsKindOf(NullValue)) in
   vs->forAll(v | v.type.oclIsKindOf( property.type))
```

6.4.17 CIMM::Qualifier

The Qualifier metaelement models qualifiers. (see 5.6.12).

Each associated value specification shall be consistent with the type of the qualifier type.

Generalization

CIMM::Element (see 6.4.5)

Attributes

No additional attributes

Associations

- The defining QualifierType

```
qualifierType : QualifierType [1]
```

- The values of the Qualifier

```
valueSpecification : ValueSpecification [0..1]
```

- The qualified element that is setting values for this qualifier

```
qualifiedElement : NamedElement [1]
```

Constraints

Constraint 6.4.17-1: A qualifier of a scalar qualifier type shall have no more than one valueSpecification

```
qualifierType.array = false implies valueSpecification->size() <= 1
```

Constraint 6.4.17-2: Values of a qualifier shall be consistent with qualifier type

```
valueSpecification->forAll(v | v.type.oclIsKindOf(qualifierType.type))
```

Constraint 6.4.17-3: The qualifier shall be applied to an element specified by qualifierType.scope

```
qualifierType.scope->includes(c | c->toUpper() = qualifiedElement.name-
>toUpper())
```

Constraint 6.4.17-4: A qualifier defined as DisableOverride shall not change its value in the propagation graph

```
qualifierType.policy=PropagationPolicyKind::disableOverride implies
(
    qualifiedElement->allOverridden()->qualifier->
          select(q | q.oclIsKindOf(Qualifier) and
             q.name->toUpper()=self.name->toUpper())->
                                  forAll(q | q.valueSpecification =
self.valueSpecification)
   and
      let fe = qualifiedelement.allOverridden()->select(f | f.allOverridden()-
>isEmpty()) in
      if fe->isEmpty() then true – self is already on the top element in the
hierarchy
      else
         let fq = fe->qualifier->select(q | q.oclIsKindOf(Qualifier) and
            q.name->toUpper()=self.name->toUpper()) in
```

```
                if (fq->size() = 1 and fq->valuespecification.value =
self.valueSpecification) then true
                else false – Error the first element is not qualified
              endif
      endif
)
```

6.4.18 CIMM::QualifierType

A QualifierType metaelement models an extension to one or more metaelements that can be applied to model elements realized from those metaelements (see 5.6.12).

Generalization

CIMM::TypedElement (see 6.4.25)

Attributes

- This enumeration defines the metaelements that are extended by as QualifierType

```
scope : QualifierScopeKind [1..*]
```

- The policy that defines the update and propagation rules for values of the qualifierType

```
policy : PropagationPolicyKind [1] = PropagationPolicyKind::enableOverride
```

Associations

- Applied qualifiers defined by this qualifier type

```
qualifier : Qualifier [0..*]
```

- The default values for qualifier types of this type.

```
defaultValue: ValueSpecification [0..1]
```

- A qualifier type belongs to a schema

```
schema: Schema[1]
```

Constraints

Constraint 6.4.18-1: If a default value is specified for a qualifier type, the value shall be consistent with the type of the qualifier type.

```
defaultValue.size()=1
implies (
   defaultValue.type.oclIsKindOf(type)
```

Constraint 6.4.18-2: The default value of a non-string qualifier type shall not be null.

```
not type.oclIsKindOf(string)
implies (
   defaultValue.size()=1 and
   not defaultValue.oclIsKindOf(NullValue)
```

Constraint 6.4.18-3: The qualifier type shall have a type that is either an enumeration, integer, string, or boolean.

```
type.oclIsKindOf(enumeration) or
type.oclIsKindOf(unlimitedNatural) or
type.oclIsKindOf(unsignedInteger) or
type.oclIsKindOf(signedInteger) or
type.oclIsKindOf(string) or
type.oclIsKindOf(boolean)
```

Operations

- The set of overridden qualifier types is always empty.

```
QualifierType:allOverridden (): Set(QualifierType);
```

```
allOverridden =  Null
```

6.4.19 CIMM::Reference

The Reference metaelement models reference properties (see 5.6.3.4).

Generalization

CIMM::Property (see 6.4.15)

Attributes

- Specifies how associated instances are aggregated. The value is determined by the AggregationType qualifier.

```
aggregationType: AggregationKind [1]
```

Associations

- No additional associations

Constraints

Constraint 6.4.19-1: The type of a reference shall be a ReferenceType

```
type.oclIsKindOf(ReferenceType)
```

Constraint 6.4.19-2: An aggregation reference in an association shall be a binary association

```
aggregationType <> AggregationKind::none implies
   structure.property->select(p | p.oclIsKindOf(Reference))->size() = 2
```

Constraint 6.4.19-3: A reference in an association shall not be an array

```
structure.oclIsKindOf(Association) implies not array
```

Constraint 6.4.19-4: A generalization of a reference shall not have a kind of its more specific type

```
subType->notEmpty() implies not self.oclIsKindOf(subType)
```

6.4.20 CIMM::ReferenceType

The ReferenceType metaelement models a reference type (see 5.6.9).

Generalization

CIMM::Type (see 6.4.24)

Attributes

No additional attributes

Associations

- The class that is referenced

```
class : Class [1]
```

Constraints

Constraint 6.4.20-1: A subclass of a ReferenceType shall refer to a subclass of the referenced Class

```
superType->notEmpty() implies class.oclIsKindOf(superType.class)
```

Constraint 6.4.20-2: ReferenceTypes are not abstract

```
not abstract
```

6.4.21 CIMM::Schema

A Schema metaelement models schemas. A schema provides a context for assigning schema unique names to the definition of elements including: associations, classes, enumerations, instance values, qualifier types, structures and structure values.

The qualifier types defined in this specification belong to a predefined schema with an empty name.

Generalization

CIMM::NamedElement (see 6.4.12)

Attributes

No additional attributes

Associations

- Types defined in this schema

```
types : Type[*]
```

- The complex values defined in this schema

```
complexValue : ComplexValue [0..*]
```

- Qualifier types defined in this schema

```
qualifierType : QualifierType[*]
```

Constraints

Constraint 6.4.21-1: All members of a schema have unique, case insensitive names.

```
Let members: Set(NamedElement) = complexValue->oclAsType(NamedElement)->
                           union(qualifierType->oclAsType(NamedElement)->
                           union(type->oclAsType(NamedElement)
in
   members = forAll(this | members->excluding(this)->
             forAll( other | this.name.toUpperCase() <>
other.name.toUpperCase() )  )
```

Operations

- The set of overridden Schemas is always empty

```
Schema:allOverridden (): Set(Schema);
    allOverridden =  Null
```

6.4.22 CIMM::Structure

A Structure metaelement models a structure (see 5.6.6).

Generalization

CIMM::Type (see 6.4.24)

Attributes

No additional attributes

Associations

- Properties owned by this structure

```
property : Property [0..*]
```

- A structure may define local structures.

```
localStructure : Structure[0..*]
```

- A structure may define local enumerations.

```
localEnumeration : Enumeration[0..*]
```

- A local structure is defined in a structure.

```
structure : Structure[0..1]
```

Constraints

Constraint 6.4.22-1: All properties of a structure have unique, case insensitive names within their structure

For details about uniqueness of property names in structures, see 5.7.2.

```
self.exposedProperties()->
   -- For each exposed property test that it does not match all others.
   forAll( memb | self.exposedProperties()->excluding(memb)->
      forAll( other | memb.name.toUpperCase() <> other.name.toUpperCase() ) )
```

Constraint 6.4.22-2: All localEnumerations of a structure have unique, case insensitive names.

For details about uniqueness of local enumeration names in structures, see 5.7.2.

```
self.exposedEnumerations()->
   -- For each exposed local enumeration test that it does not match all
others.
   forAll( memb | localEnumeration->excluding(memb)->
      forAll( other |  memb.name.toUpperCase() <> other.name.toUpperCase() )  )
```

Constraint 6.4.22-3: All localStructures of a structure have unique, case insensitive names.

For details about uniqueness of local structure names in structures, see 5.7.2.

```
self.exposedStructures()->
   -- For each exposed local structure test that it does not match all others.
   forAll( memb | self.exposedStructures()->excluding(memb)->
         forAll( other |  memb.name.toUpperCase() <> other.name.toUpperCase() )
)
```

Constraint 6.4.22-4: Local structures shall not be classes or associations

```
localStructure->forAll(c | not c.oclIsKindOf(Class))
```

Constraint 6.4.22-5: The superclass of a local structure must be schema level or a local structure within this structure’s supertype hierarchy

```
superType->notEmpty() and structure->notEmpty() implies
   superType->structure->isEmpty() -- supertype is global
   or
   exposedStructures()->includes(superType)  -- supertype is local
```

Constraint 6.4.22-6: The superclass of a local enumeration must be schema level or a local enumeration within this structure’s supertype hierarchy

```
superType->notEmpty() and enumeration->notEmpty() implies
   superType->enumeration->isEmpty() -- supertype is global
   or
   exposedEnumerations()->includes(superType)  -- supertype is local
```

Constraint 6.4.22-7: Specialization of schema level structures must be from other schema level structures

```
structure->isEmpty() and superType->notEmpty() implies superType->structure-
>isEmpty()
```

Operations

- The query allProperties() gives all of the properties in the namespace of the structure. In general, through inheritance, this will be a larger set than property.

```
Structure::allProperties(): Set(Property);
allProperties = property->union(self.allSuperTypes()->property)
```

- The exposedProperties operation excludes overridden properties.

```
Structure::exposedProperties() : Set(Property);
exposedProperties = allProperties()->
      excluding(inh | property-> select(overridden->includes(inh)))
```

- The exposedStructures operation includes all local structures in the inheritance graph.

```
Structure::exposedStructures() : Set(Structure);
exposedStructures = localStructure->union(allSuperTypes()->localStructure)
```

- The exposedEnumerations operation includes all local enumerations in the inheritance graph.

```
Enumeration::exposedEnumerations() : Set(Enumeration);
exposedEnumerations = localEnumeration->union(allSuperTypes()-
>localEnumeration)
```

6.4.23 CIMM::StructureValue

The value of a structure (see 5.6.11).

When used as the value or default value of a typed element a structure value shall not be abstract. The type of the structure value shall be the same as, or a subtype of, that element's type.

Generalization

CIMM::ComplexValue (see 6.4.4)

Attributes

No additional attributes

Associations

No additional associations

Constraints

- Constraint 6.4.23-1: A structure value is a realization of a Structure

```
type.oclIsKindOf(Structure)
```

6.4.24 CIMM::Type

A Type is an abstract metaelement that models a type (structure, class, association, primitive type, enumeration, reference type).

A Type indicates whether it is a scaler or an array.

Generalization

CIMM::NamedElement (see 6.4.12)

Attributes

- Specifies whether the model element may be realized as an instance. True indicates that the element shall not be realized. The value is determined by the Abstract qualifier.

```
abstract : boolean [1]
```

- True specifies the type is an array.

```
array : boolean[1] = false
```

- Specifies whether or not a model element may be specialized. True indicates that the element shall not be specialized. The value is determined by the Terminal qualifier.

```
terminal : boolean [1]
```

- Version is an optional string that indicates the version of the modeled type. The value is determined by Version qualifier.

```
version : string [0..1]
```

Associations

- Specifies the schema to which the type belongs

```
schema:Schema[1]
```

- Specifies a more general type; only single inheritance

```
superType : Type [0..1]
```

- Specifies the specializations of this type

```
subType : Type [0..*]
```

- Typed elements that have this type

```
typedElement : TypeElement [0..*]
```

- Values of this type

```
valueSpecification : ValueSpecification [0..*]
```

Constraints

Constraint 6.4.24-1: Terminal types shall not be abstract and shall not be subclassed

```
terminal=true implies abstract=false and subType.size()=0
```

Constraint 6.4.24-2: An instance shall not be realized from an abstract type

```
abstract implies realizedElement->isEmpty()
```

Constraint 6.4.24-3: There shall be no circular inheritance paths

```
superType->closure( t  |  t <> self )
```

Constraint 6.4.24-4: A value of an array shall be either NullValue or ArrayValue

```
array implies valueSpecification.oclIsKindOf(NullValue) or
valueSpecification.oclIsKindOf(ArrayValue)
```

Operations

- The operation allSuperTypes() gives all of the direct and indirect ancestors of a type.

```
Type::allSuperTypes(): Set(Type);  -- recursively collect supertypes
    allSuperTypes = superType->union(superType->collect(p | p.allSuperTypes())
```

- The set of overridden types is the same as the set of all supertypes.

```
Type:allOverridden (): Set(Type);
     allOverridden =  self.allSuperTypes() ))
```

6.4.25 **CIMM::TypedElement**

A TypedElement is an abstract metaelement that models typed elements. The value of a typed element shall conform to its type.

A TypedElement indicates whether or not a value is required. If no value is provided, the element is Null.

Generalization

CIMM::NamedElement (see 6.4.12)

Attributes

- Specifies the behavior of elements of an array; the value is determined by the ArrayType qualifier.
  `arrayType : CIMM::ArrayKind [1]`
- Required true specifies that elements of the type shall not be Null. The value is determined by the Required qualifier.
  `required : boolean[1]`

Associations

- Has a Type
  `type: Type [1]`

Constraints

No additional constraints

6.4.26 CIMM::ValueSpecification

A ValueSpecification is an abstract metaelement used to specify a value or values in a model.

The value specification in a model specifies a value, but shall not be in the same form as the actual value of an element in a modeled system. It is required that the type and number of values represented is suitable for the context where the value specification is used.

Values are described by the concrete subclasses of ValueSpecification. Values of primitive types are modeled in subclasses of literal value (see 6.4.9), values of enumerations are modeled using enumeration values 5.6.2), and values any other type are modeled using complex values (6.4.4).

NOTE A specific kind of value specification is used to indicate the absence of a value. In the model, this is a literal Null and is represented by the NullValue metaelement.

Generalization

CIMM::NamedElement (see 6.4.12)

Attributes

No additional attributes

Associations

- Qualifier that has this value specification
  `qualifier : Qualifier[0..1]`
- PropertySlot that has this value specification
  `propertySlot : PropertySlot[0..1]`
- An enumeration value that has this value specification
  `enumValue: EnumValue [0..1]`
- QualifierType that has this default value specification
  `qualifierType: QualifierType[0..1]`
- Parameter that has this as a default value specification
  `parameter: Parameter[0..1]`
- Property that has this default value specification
  `property: Property [0..1]`
- Type of this value
  `type: Type [1]`

- If this ValueSpecification is an element of an array, the ValueSpecification for the array

```
arrayValue: ArrayValue [0..1]
```

Constraints

Constraint 6.4.26-1: A value specification shall have one owner.

```
qualifier->size() + propertySlot->size() + enumValue->size() +
    qualifierType->size() + parameter->size() + property->size() +
    array->size() = 1
```

Constraint 6.4.26-2: A value specification owned by an array value specification shall have scalar type

```
array->notEmpty() implies type.array=false
```

- Constraint 6.4.26-3: The type of a value specification shall not be abstract

```
not type.abstract
```

7 Qualifier types

A CIM Metamodel implementation shall support the qualifier types specified by this clause.

Qualifier types and qualifiers provide a means to add metadata to schema elements (see 5.6.12).

Each qualifier adds descriptive information to the qualified element or implies an assertion that shall be true for the qualified element in a CIM Metamodel implementation. Assertions made by qualifiers should be validated along with evaluation of schema declarations. CIM Metamodel implementations shall conform to all assertions made by qualifiers. Run-time enforcement of such assertions is not required but is useful for testing purposes.

The qualifiers defined in this specification shall be specified for each CIM Metamodel implementation. Additional qualifier types may be defined.

If a qualifier type is not specified in a CIM schema implementation, then it has no effect on model elements in that implementation.

If a qualifier type is specified in a CIM schema implementation, then it conceptually adds the qualifier to all model elements that are in the scope of the qualifier type.

For a particular model element, the value of each such qualifier is as follows:

a) If it is explicitly set on that model element, then the qualifier has the value specified.
b) If the policy is disable override or enable override, and a value has been explicitly set on another model element closer to the root of its propagation graph, (see 5.6.12), then the qualifier has the nearest such value.
c) Otherwise, the qualifier has the default value if one is defined on the qualifier type or it has no value (i.e., it is Null).

NOTE The metamodel is modeling language agnostic. It is the responsibility of a modeling language definition to map the specification of qualifier types and the setting of qualifier values onto language elements. For example, there is not a means in the MOF language to directly apply a qualifierType to a method return, but because there can be at most one method return for a method, the MOF language allows specification of qualifier types that are applicable to method returns on corresponding method. Other languages could map to this metamodel more directly, for instance XMI as defined by the OMG MOF 2 XMI Mapping specification.

Unless otherwise specified, qualifier types that modify the semantics of the values of a TypedElement apply to all values of that TypedElement. Examples include BitMap, MaxSize, and PUnit.

All qualifier types defined within this clause belong to the CIM Metamodel schema.

Each qualifier type expresses a qualifier added to a set of metaelements. Set the scope to the enumeration values defined by QualifierScopeKind that correspond to those metaelements. A schema representation language must define how it maps to those enumeration values. For example, if the qualifier type affects association, class, enumeration, and structure, then:

```
scope = QualifierScopeKind::association or QualifierScopeKind::class or
      QualifierScopeKind::enumeration or QualifierScopeKind::structure
```

The policy of a qualifier type shall be set to the specified policy. For example, if the policy is specified as restricted, then:

```
policy=PropagationPolicyKind::restricted
```

The following qualifier types shall be supported by a CIM Metamodel implementation. Each clause specifies the name and semantics..

7.1 Abstract

If the value of an Abstract qualifier is true, the qualified association, class, enumeration, or structure is abstract and serves only as a base. It is not possible to create instances of abstract associations or classes, to define values of abstract structures, or to use abstract types as a type of a typed element (except for reference types).

The attributes of the qualifier type are:

```
type = boolean (scalar, non-Null)
defaultValue = false
scope = QualifierScopeKind::association or QualifierScopeKind::class or
      QualifierScopeKind::enumeration or QualifierScopeKind::structure
Policy = PropagationPolicyKind::restricted
```

Constraints

Constraint 7.1-1: The value of the Abstract qualifier shall match the abstract meta attribute

```
qualifier->forAll(q | q.valueSpecification.value=q.qualifiedElement.abstract)
```

7.2 AggregationKind

The AggregationKind qualifier shall only be specified within a binary association on a reference property, which references instances that are aggregated into the instances referenced by the other reference property.

The value of AggregationKind qualifier indicates the type of the aggregation relationship. The values are specified by the AggregationKind enumeration (see 6.3.2). A value of none indicate that the relationship is not an aggregation. Alternatively the value can indicate a shared or composite aggregation. In both of those cases, the instances referenced by the qualified property are aggregated into instances referenced by the unqualified reference property.

NOTE AggregationKind replaces the CIM v2 qualifiers Aggregate, Aggregation, and Composition. In CIM v2, Aggregation and Composition was specified on the association and the Aggregate qualifier was specified on the property that references an aggregating instance. AggregationKind is specified on the other reference property, that is the reference to an aggregated instance.

The attributes of the qualifier type are:

```
type = string (scalar, non-Null)]
defaultValue = AggregationKind::none
scope = QualifierScopeKind:: reference
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.2-1: The AggregationKind value shall be consistent with the AggregationKind attribute

```
qualifier->forAll(q | q.valueSpecification.value = q.qualifiedElement-
>asType(Reference).AggregationKind)
```

Constraint 7.2-2: The AggregationKind qualifier shall only be applied to a reference property of an Association

```
qualifier->forAll(q | q.qualifiedElement->structure.oclIsKindOf(Association))
```

7.3 ArrayType

The value of an ArrayType qualifier specifies that the qualified property, reference, parameter, or method return is an array of the specified type. The values of the ArrayType qualifier are defined by the ArrayKind enumeration (see 6.3.3).

The attributes of the qualifier type are:

```
type = string (scalar, non-Null)
defaultValue = ArrayKind::bag
scope = QualifierScopeKind::Method or QualifierScopeKind::parameter or
      QualifierScopeKind::property or QualifierScopeKind::reference
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.3-1: The ArrayType qualifier value shall be consistent with the arrayType attribute

```
qualifier->forAll( q | q.valueSpecification.value = q.qualifiedElement-
>asType(TypedElement).arrayType )
```

7.4 BitMap

The values of this qualifier specifies a set of bit positions that are significant within a method return, parameter or property having an unsigned integer type.

Bits are labeled by bit positions, with the least significant bit having a position of zero (0) and the most significant bit having the position of M, where M is one (1) less than the size of the unsigned integer type. For instance, for a uint16, M is 15.

The values of the array are unsigned integer bit positions, each represented as a string.

The position of a specific value in the Bitmap array defines an index used to select a string literal from the BitValues (see 7.5) array.

The attributes of the qualifier type are:

```
type = string (array, Null allowed)
defaultValue = Null
scope = {QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.4-1: An element qualified with Bitmap shall have type UnsignedInteger

```
qualifier.qualifiedElement->forAll(e | e.type.oclIsKindOf(UnsignedInteger))
```

Constraint 7.4-2: The number of Bitmap values shall correspond to the number of values in BitValues

```
qualifier.qualifiedElement->qualifier->select(q| q name='BitValues')->
   forAll(valueSpecification->size() = q->valueSpecification->size())
```

7.5 BitValues

The values of this qualifier specify a set of literals that corresponds to the respective bit positions specified in a corresponding BitMap qualifier type.

The position of a specific value in the Bitmap (see 7.4) array defines an index used to select a string literal from the BitValues array.

The attributes of the qualifier type are:

```
type = string (array, Null allowed)
defaultValue = Null
scope =  {QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.5-1: An element qualified by BitValues shall have type UnsignedInteger

```
qualifier.qualifiedElement->forAll(q | q.type.oclIsKindOf(UnsignedInteger))
```

Constraint 7.5-2: The number of BitValues shall correspond to the number of values in the BitMap

```
qualifier.qualifiedElement->qualifier->select(q| q name='BitMap')->
   forAll(valueSpecification->size() = q->valueSpecification->size())
```

7.6 Counter

If true, the value of a Counter qualifier asserts that the qualified element represents a counter. The type of the qualified element shall be an unsigned integer with values that monotonically increase until the value of MaxValue is reached, or until the maximum value of the datatype. At that point, the value starts increasing from the value of MinValue or zero (0), whichever is greater.

The qualifier type is specified on parameter, property, method, and qualifier type elements.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property} }
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.6-1: The element qualified by Counter shall be an unsigned integer

```
qualifier.qualifiedElement->forAll(e | e.type.oclIsKindOf(unsignedInteger))
```

Constraint 7.6-2: A Counter qualifier is mutually exclusive with the Gauge qualifier

```
qualifier.qualifiedElement->qualifier->forAll( q | not q.name->toUpper() =
'GUAGE')
```

7.7 Deprecated

A non-Null value of this qualifier indicates that the qualified element has been deprecated. The semantics of this qualifier are informational only and do not affect the element's support requirements. Deprecated means that the qualified element may be removed in the next major version of the schema following the deprecation. Replacement elements shall be specified using the syntax defined in the following ABNF:

```
replacement = ("No value" /

              (typeName *("." typeName)

                 ["." methodName ["." parameterName ] /
```

```
                  "." *(propertyName  "." ) propertyName ["."  EnumValue] ]    )
```

Where:

- The typeName rule names the ancestor Type (Association, Class, Enumeration, or Structure) that owns the replacement element.
- The methodName rule is required if the replaced element is a method. If the overridden element is a parameter, then it shall be specified.
- The propertyName rule is required if a property is replaced.

The attributes of the qualifier type are:

```
type =  string (scalar, Null allowed)
defaultValue = Null
scope ={QualifierScopeKind::any}
policy = PropagationPolicyKind::restricted
```

Constraint 7.7-1: The value of the Deprecated qualifier shall match the deprecated meta attribute

```
qualifier->forAll(q | q.valueSpecification.value=q.qualifiedElement.deprecated)
```

7.8 Description

The value of this qualifier describes the qualified element.

The attributes of the qualifier type are:

```
type =  string (scalar, Null allowed)
defaultValue = Null
scope ={QualifierScopeKind::any}
policy = PropagationPolicyKind::enableOverride
```

7.9 EmbeddedObject

If the value of this qualifier is true, the qualified string typed element contains an encoding of an instance value or an encoding of a class definition.

To reduce the parsing burden, the encoding that represents the embedded object in the string value depends on the protocol or representation used for transmitting the qualified element. This dependency makes the string value appear to vary according to the circumstances in which it is observed.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property}
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.9-1: An element qualified by EmbeddedObject shall be a string

```
qualifier->forAll(q | q.qualifiedElement.type.oclIsKindOf(string))
```

7.10 Experimental

The value of the Experimental qualifier specifies whether or not the qualified element has 'experimental' status. The implications of experimental status are specified by the organization that owns the element.

If false, the qualified element has 'final' status. Elements with 'final' status shall not be modified in backwards incompatible ways within a major schema version (see 7.28).

Experimental elements are subject to change. Elements with 'experimental' status may be modified in backwards incompatible ways in any schema version, including within a major schema version.

Experimental elements are published for developing CIM schema implementation experience. Based on CIM schema implementation experience: changes may occur to this element in future releases; the element may be standardized "as is"; or the element may be removed.

When an enumeration, structure, class, or association has the Experimental qualifier applied with a value of true, its properties, methods, literals, and local types also have 'experimental' status. In that case, it is unnecessary also to apply the Experimental qualifier to any of its local elements, and such redundant use is discouraged.

When an enumeration, structure, class, or association has 'final' status, its properties, methods, literals, and local types may individually have the Experimental qualifier applied with a value of true.

Experimental elements for which a decision is made to not take them final should be removed from their schema.

NOTE The addition or removal of the Experimental qualifier type does not require the version information to be updated.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::any}
policy = PropagationPolicyKind::restricted
```

Constraint 7.10-1: The value of the Experimental qualifier shall match the experimental meta attribute

```
qualifier->forAll(q |
q.valueSpecification.value=q.qualifiedElement.experimental)
```

7.11 Gauge

If true, the qualified integer element represents a gauge. The type of the qualified element shall be an integer with values that can increase or decrease. The value is qualified to be within the range of the elements type and within the range of any applied MinValue and MaxValue qualifier types.

The value is represented as literal boolean.

The qualifier type is specified on parameter, property, method, and qualifier type elements.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property}
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.11-1: The element qualified by Gauge shall be an unsigned integer

```
qualifier.qualifiedElement->forAll(e | e.type.oclIsKindOf(integer))
```

Constraint 7.11-2: A Counter qualifier is mutually exclusive with the Gauge qualifier

```
qualifier.qualifiedElement->qualifier->forAll( q | not q.name->toUpper() =
’COUNTER’)
```

7.12 In

If the value of an In qualifier is true, the qualified parameter is used to pass values to a method.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = true
scope ={QualifierScopeKind::parameter}
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.12-1: The value the In qualifier shall be consistent with the direction attribute

```
qualifier->forAll(q | q.valueSpecification.value = true implies
   q.qualifiedElement->asType(Parameter).direction= DirectionKind::in or
   q.qualifiedElement->asType(Parameter).direction= DirectionKind::inout )
```

7.13 IsPUnit

If the value is true, this qualifier asserts that the value of the qualified string element represents a programmatic unit of measure. The value of the string element follows the syntax for programmatic units, as defined in ANNEX D.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={ QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.13-1: The type of the element qualified by IsPUnit shall be a string.

```
qualifier.qualifiedElement->forAll(e | e.type.oclIsKindOf(string))
```

7.14 Key

If the value of a Key qualifier is true, the qualified property or reference is a key property. In the scope in which it is instantiated, a separately addressable instance of a class is identified by its class name and the name value pairs of all key properties (see 5.6.7).

The values of key properties and key references are determined once at instance creation time and shall not be modified afterwards. Properties of an array type shall not be qualified with Key. Properties qualified with EmbeddedObject or EmbeddedInstance shall not be qualified with Key. Key properties and Key references shall not be Null.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={ QualifierScopeKind::property, QualifierScopeKind::reference}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.14-1: The value of the Key qualifier shall be consistent with the key attribute

```
qualifier->forAll(q | q.valueSpecification.value = q.qualifiedElement-> key)
```

Constraint 7.14-2: If the value of the Key qualifier is true, then the value of Write shall be false

```
qualifier->forAll(q | q.qualifiedElement.key = true implies -
>q.qualifiedElement.write=false)
```

7.15 MappingStrings

Each value of this qualifier specifies that the qualified element represents a corresponding element specified in another standard. See ANNEX F for standard mapping formats.

The attributes of the qualifier type are:

```
type =  string (array, Null allowed)
defaultValue = Null
scope ={QualifierScopeKind::any}
policy = PropagationPolicyKind::enableOverride
```

7.16 Max

The value specifies the maximum size of a collection of instances referenced via the qualified reference in an association (see 5.6.8) when the values of all other references of that association are held constant. Within an instance of the containing association, the qualified reference can reference at most one instance of the collection.

If not specified, or if the qualifier type does not have a value, then the maximum is unlimited.

The attributes of the qualifier type are:

```
type =  unlimitedNatural (scalar, non-Null)
defaultValue = '*'
scope ={ QualifierScopeKind::reference }

policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.16-1: The value of the MAX qualifier shall be consistent with the value of max in the qualified element

```
qualifier->forAll(q | q.valueSpecification.value = q.qualifiedElement.max)
```

Constraint 7.16-2: MAX shall only be applied to a Reference of an Association

```
qualifier->forAll(q | q.qualifiedElement->structure.oclIsKindOf(association))
```

7.17 Min

The value specifies the minimum size of a collection of instances referenced via the qualified reference in an association (see 5.6.8) when the values of all other references of that association are held constant. Within an instance of the containing association, the qualified reference can reference at most one instance of the collection.

If not specified, or if the qualifier type does not have a value, then the minimum is zero.

The attributes of the qualifier type are:

```
type =  unsignedInteger (scalar, non-Null)
defaultValue = 0
scope = {QualifierScopeKind::reference}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.17-1: The value of the MIN qualifier shall be consistent with the value of min in the qualified element

```
qualifier->forAll(q | q.valueSpecification.value = q.qualifiedElement.min)
```

Constraint 7.17-2: MIN shall only be applied to a Reference of an Association

```
qualifier->forAll(q | q.qualifiedElement->structure.oclIsKindOf(association))
```

7.18 ModelCorrespondence

Each value of this qualifier asserts a semantic relationship between the qualified element and a named element. That correspondence should be described in the definition of those elements, but may be described elsewhere.

The format of each name value is specified by the following ABNF:

```
correspondingElementName =
        *(typeName "." )
          (methodName ["." parameterName ] /
          *(propertyName "." ) propertyName ["."  EnumValue] )
```

Where:

- The typeName rule names the ancestor Type (Association, Class, Enumeration, or Structure) that owns the corresponding element and is required if an element of the same name is exposed more than once in the ancestry.
- The methodName rule is required if the overridden element is a method. If the overridden element is a parameter, then it shall be specified.
- The propertyName rule is required if a property is overridden.

The basic relationship between the referenced elements is a "loose" correspondence, which simply indicates that the elements are coupled. This coupling may be unidirectional. Additional meta information may be used to describe a tighter coupling.

The following list provides examples of several correspondences:

- A property provides more information for another. For example, an enumeration has an allowed value of "Other", and another property further clarifies the intended meaning of "Other." In another case, a property specifies status and another property provides human-readable strings (using an array construct) expanding on this status. In these cases, ModelCorrespondence is found on both properties, each referencing the other.
- A property is defined in a subclass to supplement the meaning of an inherited property. In this case, the ModelCorrespondence is found only on the construct in the subclass.
- Multiple properties taken together are needed for complete semantics. For example, one property could define units, another property could define a multiplier, and another property could define a specific value. In this case, ModelCorrespondence is found on all related properties, each referencing all the others.
  NOTE This specification supports structures. A structure implies a relationship between its properties.
- Multiple related arrays are used to model a multi-dimensional array. For example, one array could define names while another defines the name formats. In this case, the arrays are each defined with the ModelCorrespondence qualifier type, referencing the other array properties or parameters. Also, they are indexed and they carry the ArrayType qualifier type with the value "Indexed."
  NOTE This specification supports structures. A structure implies a relationship between its properties. Properties that have type structure could be arrays.

The semantics of the correspondence are based on the elements themselves. ModelCorrespondence is only a hint or indicator of a relationship between the elements.

While they do not replace all uses of ModelCorrespondence:

- structures should be used in new schemas to gather indexed array properties belonging to the same type (i.e., association, class, or structure).

- OCL constraints should be used when the correspondence between elements can be expressed as an OCL expression.

The attributes of the qualifier type are:

```
type =  string (array, Null allowed)
defaultValue = Null
scope ={QualifierScopeKind::any}
policy = PropagationPolicyKind::enableOverride
```

7.18.1 Referencing model elements within a schema

The ability to reference specific elements of a schema from other elements within a schema is required. Examples of elements that reference other elements are: the MODELCORRESPONDENCE and OCL qualifier types. This clause defines common naming rules.

- Schema.

```
schemaName = IDENTIFIER
```

- Class, Association, Structure.

```
className = [[ schemaName ] "_"]IDENTIFIER

structureName = [[ schemaName ] "_"]IDENTIFIER

qualifiedStructureName = className / structureName)

                            *("." structureName)
```

- Enumeration

```
enumerationName = [[ schemaName ] "_" ] IDENTIFIER

qualifiedEnumName = [qualifiedStructureName "."]enumerationName
```

- Property

```
propertyName = IDENTIFIER

qualifiedPropertyName = [qualifiedStructureName "."]

              propertyName *("." propertyName)
```

- Method

```
methodName = IDENTIFIER

qualifiedMethodName = [className "."] methodName
```

- Parameter

```
parameterName = IDENTIFIER

qualifiedParmName = [qualifiedMethodName "."]

              parameterName *("." propertyName)
```

- EnumValue

```
EnumValue = IDENTIFIER

qualifiedEnumValue = [qualifiedEnumName "."] EnumValue
```

- QualifierType

```
qualifierType = [ schemaName ] "_" IDENTIFIER
```

7.19 OCL

Values of this qualifier each specify an OCL statement on the qualified element.

Each OCL qualifier has zero (0) or more literal strings that each hold the value of one OCL statement, (see clause 8).

The context (i.e., self) of a specified OCL statement is the qualified element. All names used in an OCL statement shall be local to that element.

The attributes of the qualifier type are:

```
type =  string (array, Null allowed)
defaultValue = Null
scope ={QualifierScopeKind::association,  QualifierScopeKind::class,
   QualifierScopeKind::structure,
   QualifierScopeKind::method}
policy = PropagationPolicyKind::enableOverride
```

7.20 Out

If the value of an Out qualifier is true, the qualified parameter is used to pass values out of a method.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::parameter}
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.20-1: The value of the Out qualifier shall be consistent with the direction attribute

```
qualifier->forAll(q | q.valueSpecification.value = true implies
   q.qualifiedElement->asType(Parameter).direction= DirectionKind::out or
   q.qualifiedElement->asType(Parameter).direction= DirectionKind::inout )
```

7.21 Override

If the value of an Override qualifier is true, the qualified element is merged with the inherited element of the same name in the ancestry of the containing type (association, class, or structure). The qualified element replaces the inherited element.

The ancestry of an element is the set of elements that results from recursively determining its ancestor elements. An element is not considered part of its ancestry.

The ancestor of an element depends on the kind of element, as follows:

- For a class or association, its superclass is its ancestor element. If the class or association does not have a superclass, it has no ancestor.
- For a structure, its supertype is its ancestor element. If the structure does not have a supertype, it has no ancestor.
- For an overriding property (including references) or method, the overridden element is its ancestor. If the property or method is not overriding another element, it does not have an ancestor.

- For a parameter of an overriding method, the like-named parameter of the overridden method is its ancestor. If the method is not overriding another method, its parameters do not have an ancestor.

The merged element is inherited by subtypes of the type that contains the qualified element.

NOTE This qualifier type is defined as ‘restricted’. This means that if the qualified element is again specified in a subtype within the inheritance hierarchy, the qualified element will not be merged with the new descendant element unless the Override qualifier is also specified on the new descendant.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::property, QualifierScopeKind::Method,
QualifierScopeKind::parameter}
policy = PropagationPolicyKind::restricted
```

7.22 PackagePath

A package is a namespace for class, association, structure, enumeration, and package elements. That is, all elements belonging to the same package shall have unique names. Packages may be nested and are used to organize elements of a model as defined in UML (see the Unified Modeling Language: Superstructure specification).

The value of this qualifier specifies a schema relative name for a package. If a value is not specified, or is specified as Null, the package path shall be the schema name of the qualified element, followed by "::default". The format of the value for a PackagePath conforms to the following ABNF:

```
schemaName = IDENTIFIER

packageName = IDENTIFIER

packagePath = SchemaName "::"

               ("default" / packageName *("::" packageName))
```

Example 1: Consider a class named "ACME_Abc" that is in a package named "PackageB" that is in a package named "PackageA" that, in turn, is in a package named "ACME". The resulting qualifier type value for this class is "ACME::PackageA::PackageB"

Example 2: Consider a class named "ACME_Xyz" with no PackagePath qualifier type. The resulting qualifier type value for this class is "ACME::default".

The attributes of the qualifier type are:

```
type =  string (scalar, Null allowed)
defaultValue = Null
scope ={ QualifierScopeKind::association,  QualifierScopeKind::class,
QualifierScopeKind::enumeration,
   QualifierScopeKind::structure}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.22-1: The name of all qualified elements having the same PackagePath value shall be unique.

```
Let pkgNames : Set(String) = qualifier->valueSpecification.value->asSet() in
   Sequence(1..pkgNames.size())->forAll(i |
      let pkgQualifiers : Set(qualifier) =
         qualifier->select(q | q.valueSpecification.value = pkgName.at(i)) in
            Sequence(1..pkgQualifiers.size())->
               forAll(pq | pkgQualifiers.at(pq)->qualifiedElement->isUnique(e
| e.name)
```

7.23 PUnit

If the value of this qualifier is not Null, the value of the qualified numeric element is in the specified programmatic unit of measure. The specified value of the PUnit qualifier conforms to the syntax for programmatic units is defined in ANNEX D.

NOTE String typed schema elements that are used to represent numeric values in a string format cannot have the PUnit qualifier type specified, because the reason for using string typed elements to represent numeric values is typically that the type of value changes over time, and hence a programmatic unit for the element needs to be able to change along with the type of value. This can be achieved with a companion schema element whose value specifies the programmatic unit in case the first schema element holds a numeric value. This companion schema element would be string typed and the IsPUnit meta attribute would be set to true.

The attributes of the qualifier type are:

```
type = string (scalar, Null allowed)
defaultValue = Null
scope ={ QualifierScopeKind::property,
       QualifierScopeKind::Method,  QualifierScopeKind::parameter}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.23-1: The type of the element qualified by PUnit shall be a Numeric

```
type.oclIsKindOf(Numeric)
```

7.24 Read

If the value of this qualifier is true, the qualified property can be read.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = true
scope ={QualifierScopeKind::property, QualifierScopeKind::reference }
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.24-1: The value of the Read qualifier shall be consistent with the accessibility attribute

```
qualifier->forAll(q | q.valueSpecification.value = true implies
   q.qualifiedElement->asType(Property).accessibility= AccessKind::readonly or
   q.qualifiedElement->asType(Property). accessibility = AccessKind::readwrite
)
```

7.25 Required

If the value of a Required qualifier is true then: a qualified property or reference shall not be Null within a separately addressable instance of a class containing that element; and a qualified parameter shall not be Null when passed into or out of a method; and a method return shall not be Null when returned from a passed out of a method.

For an element that is an array, required does not prohibit individual elements from being Null. Table 14 and Table 15 show the consequences of setting required to true on scalar and array elements.

Table 14 – Required as applied to scalars

Required	Element value
False	Null is allowed
True	Null is not allowed

Table 15 – Required as applied to arrays

Required	Array has Elements	Array value	Array element values
False	No	Null is allowed	Not Applicable
False	Yes	Not Null	May be Null
True	No	Null is not allowed	Not Applicable
True	Yes	Not Null	May be Null

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::Method, QualifierScopeKind::parameter,
QualifierScopeKind::property,
      QualifierScopeKind::reference}
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.25-1: The value of the Required qualifier shall be consistent with the required attribute

```
qualifier->forAll(q | q.valueSpecification.value = q.qualifiedElement-
>asType(TypedElement).required)
```

7.26 Static

If the value of a Static qualifier is true, the qualified method is static (see 6.4.10).

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={ QualifierScopeKind::method }
policy = PropagationPolicyKind::disableOverride
```

Constraints

Constraint 7.26-1: The value of the Static qualifier shall be consistent with the static attribute

```
qualifier->forAll(q |
   (q.qualifiedElement.oclIsKindOf(Property) implies
       q.valueSpecification.value = q.qualifiedElement-
>asType(Property).static) or
   (q.qualifiedElement.oclIsKindOf(Method) implies
       q.valueSpecification.value = q.qualifiedElement->asType(Method).static)
)
```

7.27 Terminal

If true, the value of the Terminal qualifier specifies that the qualified element shall not have sub types.

The qualified element shall not be abstract.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = false
scope ={QualifierScopeKind::association,  QualifierScopeKind::class,
QualifierScopeKind::enumeration,
   QualifierScopeKind::structure}
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.27-1: The element qualified by Terminal qualifier shall not be abstract

```
qualifier.qualifiedElement->forAll(e | e.abstract=false)
```

Constraint 7.27-2: The element qualified by Terminal qualifier shall not have subclasses

```
qualifier.qualifiedElement->forAll(e | e.subType->isEmpty())
```

7.28 Version

The value of this qualifier specifies the version of the qualified element. The version increments when changes are made to the element.

NOTE Starting with CIM Schema 2.7 (including extension schema), the Version qualifier type shall be present on each class to indicate the version of the last update to the class.

The string representing the version comprises three decimal integers separated by periods; that is, Major.Minor.Update, as defined the versionFormat ABNF rule (see A.3).

NOTE A version change applies only to elements that are local to the class. In other words, the version change of a superclass does not require the version in the subclass to be updated.

The version shall be updated if the Experimental qualifier value is changed.

NOTE The version is updated for changes of the Experimental qualifier to enable tracking that change.

The attributes the Version qualifier type are:

```
type =  string (scalar, Null allowed)
defaultValue = Null
scope ={ QualifierScopeKind::association, QualifierScopeKind::class,
QualifierScopeKind::enumeration, QualifierScopeKind::structure}
policy = PropagationPolicyKind::restricted
```

Constraints

Constraint 7.28-1: The value of the Version qualifier shall be consistent with the version of the qualified element

```
qualifier->forAll(q | q.qualifiedElement.version = q.valueSpecification.value)
```

7.29 Write

If the value of this qualifier is true, the qualified property can be written.

The attributes of the qualifier type are:

```
type =  boolean (scalar, non-Null)
defaultValue = Null
scope ={QualifierScopeKind::property, QualifierScopeKind::reference }
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.29-1: The value of the Write must be consistent with the accessibility attribute

```
qualifier->forAll(q | q.valueSpecification.value = true implies
   q.qualifiedElement->asType(Property).accessibility= AccessKind::writeonly or
   q.qualifiedElement->asType(Property). accessibility = AccessKind::readwrite
)
```

7.30 XMLNamespaceName

If the value of this qualifier is not Null, then the value shall identify an XML schema and this qualifier asserts that values of the qualified element conforms to the specified XML schema.

The value of the qualifier is a string set to the URI of an XML schema that defines the format of the XML instance document that is the value of the qualified string element.

As defined in NamingContexts in XML, the format of the XML Namespace name shall be that of a URI reference as defined in RFC3986. Two such URI references can be equivalent even if they are not equal according to a character-by-character comparison (e.g., due to usage of URI escape characters or different lexical case).

If the value of the XMLNamespaceName qualifier type overrides an XMLNamespaceName qualifier type specified on an ancestor of the qualified element, the XML schema specified on the qualified element shall be a subset or restriction of the XML schema specified on the ancestor element, such that any XML instance document that conforms to the XML schema specified on the qualified element also conforms to the XML schema specified on the ancestor element.

No particular XML schema description language (e.g., W3C XML Schema as defined in XML Schema Part 0: Primer Second Edition or RELAX NG as defined in ISO/IEC 19757-2) is implied by usage of this qualifier.

The attributes of the qualifier type are:

```
type =  string (scalar, Null allowed)
defaultValue = Null
scope ={ QualifierScopeKind::parameter, QualifierScopeKind::property,
QualifierScopeKind::Method }
policy = PropagationPolicyKind::enableOverride
```

Constraints

Constraint 7.30-1: An element qualified by XMLNamespaceName shall be a string

```
qualifier->qualifiedElement->forAll(e | e.type.oclIsKindOf(string)
```

8 Object Constraint Language (OCL)

The Object Constraint Language (OCL) is a formal language for the description of constraints on the use of model elements. For example, OCL constraints specified against an element of a metamodel affect the use of that metaelement to construct elements of a model. Similarly, constraints specified against an element of a user model affect all instances of that element.

Examples in this clause are drawn from elements in the CIM Metamodel. However, OCL can be used on the elements of any model. The OCL qualifier provides a mechanism to specify constraints in a user model.

OCL is defined by the Open Management Group (OMG) in the Object Constraint Language specification, which states that OCL is intended as a specification language. Some OCL query functions included in this subset are defined in the UML Superstructure Specification.

OCL expressions do not change anything in a model, but rather are intended to evaluate whether or not a modeled system is conformant to a specification. This means that the state of the system will never change because of the evaluation of an OCL expression.

This specification uses a subset of OCL to specify constraints on the metaelements of clause 6 and on the use of qualifiers defined by Qualifier Types specified in this clause. Additionally, the subset described here is intended to specify the subset of OCL that shall be supported for use with the OCL qualifier.

8.1 Context

Each OCL statement is made in the context of a model element that provides for naming uniqueness. The keyword "self" is an explicit reference to that context element. All other model elements referenced in a constraint are named relative to the context element. In most expressions, "self" does not need to be explicitly stated. For example, if CIMM::NamedElement is the context, then "self.name" and "name" both refer to the name attribute of CIMM::NamedElement.

An OCL qualifier may be specified on any element. The context for evaluation of the specified OCL statements is the containing structure, class, association, enumeration, or method. For example, consider a class with a method "ShutDown", which has a boolean parameter "Force". To specify that "Force" must always be true when the method is invoked, the following OCL constraint can be specified on the method:

```
pre: force=true
```

If, instead, the same constraint was associated with the class, that OCL constraint would have to be specified as:

```
pre: shutdown::force=true
```

Associating the constraint with the class can be advantageous if it is desired to specify that "Force" must be true when some property of the class has a particular value. In the example below, if the class "State" property has value of "disaster", then the "Force" argument must be true:

```
pre: state="disaster" implies shutdown::force=true
```

8.2 Type conformance

OCL uses a type system that maps onto the types defined by CIM as defined in Table 16.

Table 16 – OCL and CIM Metamodel types

OCL Type	CIM Metamodel Type	Example
Boolean	Boolean	true, false
Integer	Integer	0, 15, -23, …
Real	Real	1.5, 0.47, …
String	String	"OCL is useful in CIM"
Enumeration	Enumeration	Blue, Green, Yellow
UML Classifiers	Types	NamedElement, Property, Class …

Collection, Set, Bag, Sequence, and Tuple are basic OCL types as well.

Specific rules for all OCL types are defined in the Object Constraint Language specification.

8.3 Navigation across associations

OCL allows traversing associations in both directions, regardless of whether or not the reference properties are owned by an association or by an associated class. While CIM Metamodel only uses associations where the reference properties are owned by the associated classes, both styles are supported by CIM Metamodel conformant models.

NOTE To simplify the conformance checking of CIM Metamodel conformant models, all associations in the CIM Metamodel are owned by the associated classes.

Starting at one class in a model, typical OCL navigation follows a referencing property to an associated class. However when association classes are used, which is common in user models, such referencing properties do not exist in the associated classes. This is resolved by first referencing the association class name, and then following the reference property in that association class. This strategy is described in the Object Constraint Language specification in its sections 7.6.4 "Navigation to Association Classes" and 7.6.5 "Navigation from Association Classes". In that specification, the reference properties in association classes are referred to as "roles".

As an example, (see Figure 2 – Example schema), from the point of view of a GOLF_Club, the role professional is ambiguous. This is solved in OCL by including the name of the association class. For example, in the context of a GOLF_Club, the following invariant asserts that there must at least one GOLF_Professional on staff.

```
Inv: GOLF_ProfessionalStaffMember.size() > 0
```

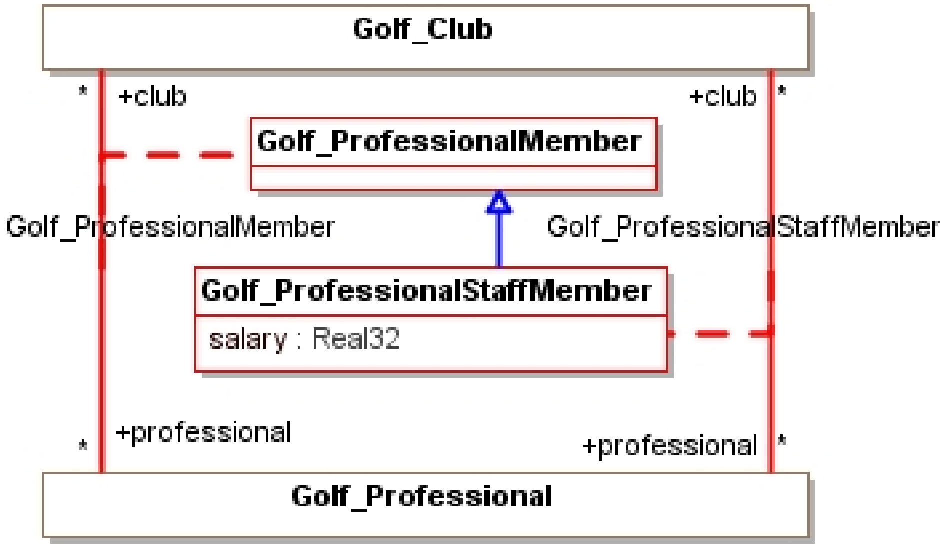

Figure 2 – Example schema

8.4 OCL expressions

The OCL specification provides syntax for creating expressions that produce an outcome of a specific type. The following subsections specify those aspects of OCL expressions that CIM Metamodel depends on. These are referred in subsequent ABNF as `OCLExpression`.

8.4.1 Operations and precedence

Table 17 lists the operations in order of precedence.

Table 17 – Operations

Operator	Description
"(", ")"	Encapsulate and de-encapsulate operations All operations within an encapsulation are evaluated before any values outside of an encapsulation.
".", "->"	Dot operations take the value, and arrow operations dereference
"not", "-"	Logical not and arithmetic negative operations
"*", "/"	Multiplication and division operations
"+", "-"	Addition and Subtraction operations
"if-then-else-endif"	Conditional execution
"<", ">", "<=", ">="	Comparison operations
"=", "<>"	Equality operations
"and"	Logical boolean conjunction operation
"or"	Logical boolean disjunction operation
"xor"	Logical boolean exclusive disjunction (exclusive or) operation
"implies"	If this is true, then this other thing must be true
"let-in"	Define a variable and use it in the following

8.4.2 OCL expression keywords

The following are OCL reserved words.

Table 18 – OCL expression keywords

and	def	derive	else	endif	if	implies	in	init
inv	let	not	or	post	pre	then	xor	

8.4.3 OCL operations

Table 19, Table 20, and Table 21 list OCL operations used by this specification or recommended for use in CIM Metamodel conformant models.

2439 **Table 19 – OCL operations on types**

Operation	Result
oclAsType()	Casts self to the specified type if it is in the hierarchy of self or undefined.
oclIsKindOf()	True if self is a kind of the specified type.
oclIsUndefined()	True if the result is undefined or Null.

2440 **Table 20 – OCL operations on collections**

Operation	Result
asSet()	Converts a Bag or Sequence to a Set (duplicates are removed.)
at()	The nth element of an Ordered Set or a Sequence (Note: A string is treated as a Sequence of characters.)
closure()	Like select, but closure returns results from the elements of a collection, the elements of the elements of a collection, the elements of the elements of the elements of a collection, and so forth
collect()	A derived collection of elements
count()	The number of times a specified object occurs in collection
excludes()	True if the specified object is not an element of collection
excluding()	The set containing all the elements in a collection except for the specified element(s)
exists()	True if the expression evaluates to true for at least one element in a source collection
forAll()	True if the expression evaluates to true for every element in a source collection
includes()	True if the specified object is an element of collection
includesAll()	True if self contains all of the elements in the specified collection
isEmpty()	True if self is the empty collection
notEmpty()	True if self is not the empty collection
sequence{}	
select()	The subset of elements from the a source collection for which the expression evaluates to true
size()	The number of elements in a collection NOTE 1 OCL coerces a Null to an empty collection. NOTE 2 OCL does not coerce a scalar to a collection.
union()	The union of the collection with another collection

2441 **Table 21 – OCL operations on strings**

Operation	Result
concat()	The specified string appended to the end of self
substring()	The substring of self, starting at a first character number and including all characters up to a second character number Character numbers run from 1 to self.size().
toUpperCase()	Converts self to upper case, if appropriate to the locale; otherwise, returns the same string as self

8.4.3.1 Let expressions

The let expression allows a variable to be defined and used multiple times within an OCL constraint.

```
letExpression = "let" varName ":" typeName "=" varInitializer "in"
                    oclExpression
```

- varName is a name for a variable.
- typeName is the type of the variable.
- varInitializer is the OCL statement that evaluates to a typeName conformant value.
- oclExpression is an OCL statement that utilizes varName.

8.5 OCL statement

The following sub clauses define the subset of OCL used by this document and which shall be supported by the OCL qualifier.

By default, ABNF rules (including literals) are to be assembled without inserting any additional whitespace characters, consistent with RFC5234. If an ABNF rule states "whitespace allowed", zero or more of the following whitespace characters are allowed between any ABNF rules (including literals) that are to be assembled:

- U+0009 (horizontal tab)
- U+000A (linefeed, newline)
- U+000C (form feed)
- U+000D (carriage return)
- U+0020 (space)

The value for a single OCL constraint is specified by the following ABNF:

```
oclStatement  = *ocl_comment          ;8.5.1
                (oclDefinition /      ;8.5.2
                oclInvariant /        ;8.5.3
                oclPrecondition /     ;8.5.4
                oclPostcondition /    ;8.5.5
                oclBodycondition /    ;8.5.6
                oclDerivation /       ;8.5.7
                oclInitialization)    ;8.5.8
```

8.5.1 Comment statement

Comments in OCL are written using either of two techniques:

- The line comment starts with the string ‘--’ and ends with the next newline.
- The paragraph comment starts with the string ‘/*’ and ends with the string ‘*/.’ Paragraph comments may be nested.

8.5.2 OCL definition statement

OCL definition statements define OCL attributes and OCL operations that are reusable by other OCL statements.

The attributes and operations defined by OCL definition statements shall be available to all other OCL statements within the its context.

A value specifying an OCL definition statement shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclDefinition = "def" [ocl_name] ":" oclExpression
```

- ocl_name is a name by which the defined attribute or operation can be referenced.
- oclExpression is the specification of the definition statement, which defines the reusable attribute or operation.

NOTE The use of the OCL keyword *self* to scope a reference to a property is optional.

8.5.3 OCL invariant constraints

OCL invariant constraints specify a boolean expression that shall be true for the lifetime of an instance of the qualified class or association.

A value specifying an OCL definition invariant constraint shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclInvariant = "inv" [ocl_name] ":" oclExpression
```

- ocl_name is a name by which the invariant expression can be referenced.
- oclExpression is the specification of the invariant constraint, which defines the boolean expression.

NOTE The use of the OCL keyword *self* to scope a reference to a property is optional.

8.5.4 OCL precondition constraint

An OCL precondition constraint is expressed as a typed OCL expression that specifies whether the precondition is satisfied. The type of the expression shall be boolean. For the method to be completed successfully, all preconditions of a method shall be satisfied before it is invoked.

A string value specifying an OCL precondition constraint shall conform to the formal syntax defined in ABNF (whitespace allowed):

```
oclPrecondition = "pre" [ocl_name] ":" oclExpression
```

- ocl_name is the name of the OCL constraint.
- oclExpression is the specification of the precondition constraint, which defines the boolean expression.

8.5.5 OCL postcondition constraint

An OCL postcondition constraint is expressed as a typed OCL expression that specifies whether the postcondition is satisfied. The type of the expression shall be boolean. All postconditions of the method shall be satisfied immediately after successful completion of the method.

A string value specifying an OCL post-condition constraint shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclPostcondition = "post" [ocl_name] ":" oclExpression
```

- ocl_name is the name of the OCL constraint.
- oclExpression is the specification of the post-condition constraint, which defines the boolean expression.

8.5.6 OCL body constraint

An OCL body constraint is expressed as a typed OCL expression that specifies the return value of a method. The type of the expression shall conform to the CIM datatype of the return value. Upon successful completion, the return value of the method shall conform to the OCL expression.

A string value specifying an OCL body constraint shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclBodycondition = "body" [ocl_name] ":" oclExpression
```

- ocl_name is the name of the OCL constraint.
- oclExpression is the specification of the body constraint, which defines the method return value.

8.5.7 OCL derivation constraint

An OCL derivation constraint specifies the derived value for a property at any time in the lifetime of the instance. The type of the expression shall conform to the CIM datatype of the property.

A string value specifying an OCL derivation constraint shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclDerivation = "derive" ":" oclExpression
```

- oclExpression is the specification of the derivation constraint, which defines the typed expression.

8.5.8 OCL initialization constraint

An OCL initialization constraint is expressed as a typed OCL expression that specifies the initial value for a property. The type of the expression shall conform to the CIM datatype of the property.

A string value specifying an OCL initialization constraint shall conform to the following formal syntax defined in ABNF (whitespace allowed):

```
oclInitialization =  "init" ":" oclExpression
```

- oclExpression is the specification of the initialization constraint, which defines the typed expression.

8.6 OCL constraint examples

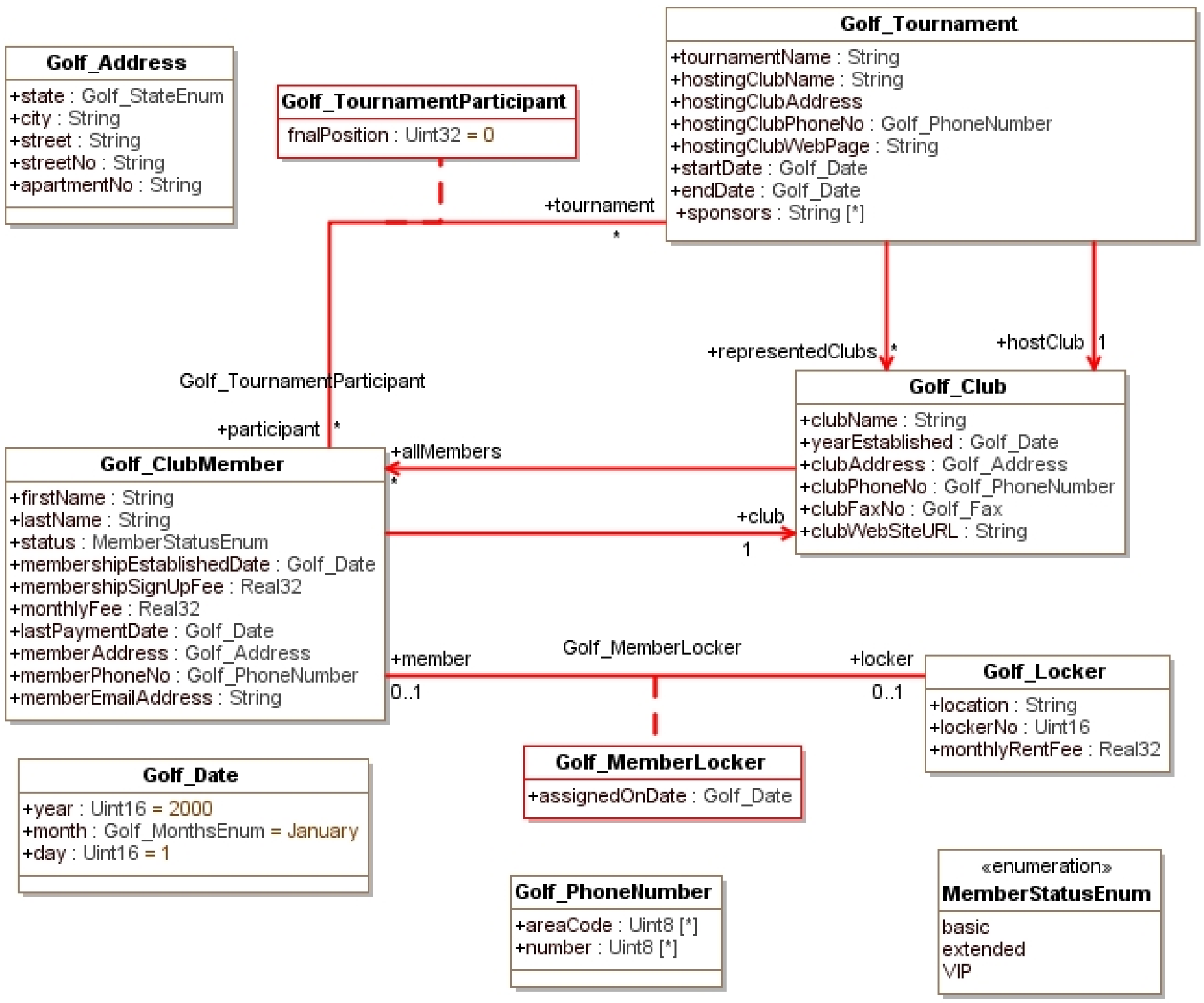

Figure 3 – OCL constraint example

The following examples refer to Figure 3 – OCL constraint example.

EXAMPLE 1: Check that property firstName and property lastName cannot be both be Null in any instance of GOLF_ClubMember. Define OCL constraint on GOLF_ClubMember as:

```
inv:      not (firstName=Null and lastName=Null)
```

EXAMPLE 2: Derive the monthly rental as 10% of the member's monthly fee. We know from the UML diagram that GOLF Locker is associated with at most one GOLF_ClubMember via the member role of the GOLF_MemberLocker association. Define OCL constraint on Golf_Locker as:

```
derive:   if GOLF_MemberLocker.member->notEmpty()
              then monthlyRentFee = GOLF_MemberLocker.member.monthlyFee * .10
              else monthlyRentFee = 0
              endif
```

EXAMPLE 3: From GOLF_ClubMember, assert that a member with basic status is permitted to have only one locker:

```
inv:   status = MemberStatusEnum.basic implies
                 not ( GOLF_MemberLocker.locker -> size() > 1 )
```

EXAMPLE 4: From GOLF_ClubMember, assert that a member must have a defined phone number:

```
Inv:   not memberPhoneNo.oclIsUndefined()
```

EXAMPLE 5: From GOLF_Tournament, assert that a member must belong to a club in the tournament:

```
-- each participant must belong to a represented club
GOLF_TournamentParticipant.participant->forAll( p | representedClubs->includes(
p.club ) )
```

EXAMPLE 6: From GOLF_Tournament, assert that hostClub refers to exactly one club.

```
hostClub.size()=1
```

ANNEX A
(normative)
Common ABNF rules

A.1 Identifiers

The following ABNF is used for element naming throughout this specification.

```
DIGIT = U+0030-0039           ; "0" ... "9"
UNDERSCORE = U+005F              ; "_"
LOWERALPHA = U+0061-007A      ; "a" ... "z"
UPPERALPHA = U+0041-005A      ; "A" ... "Z"
firstIdentifierChar = UPPERALPHA / LOWERALPHA / UNDERSCORE
nextIdentifierChar = firstIdentifierChar / DIGIT
IDENTIFIER = firstIdentifierChar *( nextIdentifierChar )
```

A.2 Integers

No whitespace is allowed in the following ABNF Rules.

```
positiveDecimalDigit = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
decimalDigit = "0" / positiveDecimalDigit
integerValue = 1*decimalDigit
positiveIntegerValue = positiveDecimalDigit *decimalDigit
decimalValue =[ "+" / "-" ]
              ( positiveDecimalDigit *decimalDigit / "0" )
```

A.3 Version

The version is represented as a string that comprises three unsigned integers separated by periods, major.minor.update, as defined by integerValue ABNF rule (see A.2) and the following ABNF:

No whitespace is allowed in the following ABNF Rules.

```
major = integerValue
minor = integerValue
update = integerValue
versionFormat = major ["." minor ["." update]]
```

EXAMPLE

```
version = "3.0.0"
version = "1.0.1
```

ANNEX B
(normative)
UCS and Unicode

ISO/IEC 10646 defines the Universal Coded Character Set (UCS). The Unicode Standard defines Unicode. This clause gives a short overview on UCS and Unicode for the scope of this document, and defines which of these standards is used by this document.

Even though these two standards define slightly different terminology, they are consistent in the overlapping area of their scopes. Particularly, there are matching releases of these two standards that define the same UCS/Unicode character repertoire. In addition, each of these standards covers some scope that the other does not.

This document uses ISO/IEC 10646 and its terminology. ISO/IEC 10646 references some annexes of The Unicode Standard. Where it improves the understanding, this document also states terms defined in The Unicode Standard in parenthesis.

Both standards define two layers of mapping:

- Characters (Unicode Standard: abstract characters) are assigned to UCS code positions (Unicode Standard: code points) in the value space of the integers 0 to 0x10FFFF.

  In this document, these code positions are referenced using the U+ format defined in ISO/IEC 10646. In that format, the aforementioned value space would be stated as U+0000 to U+10FFFF.

  Not all UCS code positions are assigned to characters; some code positions have a special purpose and most code positions are available for future assignment by the standard.

  For some characters, there are multiple ways to represent them at the level of code positions. For example, the character "LATIN SMALL LETTER A WITH GRAVE" (à) can be represented as a single *pre-composed character* at code position U+00E0 (à), or as a sequence of two characters: A *base character* at code position U+0061 (a), followed by a *combination character* at code position U+0300 (`). ISO/IEC 10646 references The Unicode Standard, Annex #15: Unicode Normalization Forms for the definition of *normalization forms*. That annex defines four normalization forms, each of which reduces such multiple ways for representing characters in the UCS code position space to a single and thus predictable way. The Character Model for the World Wide Web 1.0: Normalization recommends using *Normalization Form C* (NFC) defined in that annex for all content, because this form avoids potential interoperability problems arising from the use of canonically equivalent, yet differently represented, character sequences in document formats on the Web. NFC uses pre-composed characters where possible, but not all characters of the UCS character repertoire can be represented as pre-composed characters.

- UCS code position values are assigned to binary data values of a certain size that can be stored in computer memory.

  The set of rules governing the assignment of a set of UCS code points to a set of binary data values is called a *coded representation form* (Unicode Standard: *encoding form*). Examples are UCS-2, UTF-16 or UTF-8.

Two sequences of binary data values representing UCS characters that use the same normalization form and the same coded representation form can be compared for equality of the characters by performing a binary (e.g., octet-wise) comparison for equality.

ANNEX C
(normative)
Comparison of values

This annex defines comparison of values for equality and ordering.

Values of boolean datatypes shall be compared for equality and ordering as if "true" was 1 and "false" was 0 and the mathematical comparison rules for integer numbers were used on those values.

Comparison is supported between all numeric types. When comparisons are made between different numeric types, comparison is performed using the type with the greater precision.

Values of integer number datatypes shall be compared for equality and ordering according to the mathematical comparison rules for the integer numbers they represent.

Values of real number datatypes shall be compared for equality and ordering according to the rules defined in ANSI/IEEE 754.

Values of the string datatypes shall be compared for equality on a UCS character basis, by using the string identity matching rules defined in chapter 4 "String Identity Matching" of the Character Model for the World Wide Web 1.0: Normalization specification.

In order to minimize the processing involved in UCS normalization, string typed values should be stored and transmitted in Normalization Form C (NFC) as defined in The Unicode Standard, Annex #15: Unicode Normalization Forms. This allows skipping the costly normalization when comparing the strings.

This document does not define an order between values of the string datatypes, since UCS ordering rules could be compute intensive and their usage can be decided on a case by case basis. The ordering of the "Common Template Table" defined in ISO/IEC 14651 provides a reasonable default ordering of UCS strings for human consumption. However, an ordering based on the UCS code positions, or even based on the octets of a particular UCS coded representation form is typically less compute intensive and might be sufficient, for example when no human consumption of the ordering result is needed.

Two values of the octetstring datatype shall be considered equal if they contain the same number of octets and have equal octets in each octet pair in the sequences. An octet sequence S1 shall be considered less than an octet sequence S2, if the first pair of different octets, reading from left to right, is beyond the end of S1 or has an octet in S1 that is less than the octet in S2. This comparison rule yields the same results as the comparison rule defined for the strcmp() function in IEEE Std 1003.1.

Two values of the reference datatype shall be considered equal if they resolve to the same instance in the same QualifiedElement. This document does not define an order between two values of the reference datatype.

Two values of the datetime datatype shall be compared based on the time interval or point in time they represent, according to mathematical comparison rules for these numbers. As a result, two datetime values that represent the same point in time using different time zone offsets are considered equal.

Two values of compatible datatypes that both have no value, (i.e., are Null), shall be considered equal. This document does not define an order between two values of compatible datatypes where one has a value, and the other does not.

Two array values of compatible datatypes shall be considered equal if they contain the same number of array entries and in each pair of array entries, the two array entries are equal. This document does not define an order between two array values.

Two structure or instance values shall be considered equal if they have the same type and if all properties with matching names compare as equal.

ANNEX D
(normative)
Programmatic units

This annex defines the concept of a *programmatic unit* and a syntax for representing programmatic units as strings.

A programmatic unit is an expression of a unit of measurement for programmatic access. The goal is that programs can make sense of a programmatic unit by parsing its string representation, and can perform operations such as transformations into other (compatible) units, or combining multiple programmatic units. The string representation of programmatic units is not optimized for use in human interfaces.

Programmatic units can be used as a value of the PUnit qualifierType, or as a value of any string typed schema element whose values represents a unit. The boolean IsPUnit qualifierType can be used on a string typed schema element to declare that its value is a string representation of a programmatic unit.

A programmatic unit can be as simple as a single base unit (for example, "byte"), or in the most complex cases can consist of a number of base units and numerical multipliers (including standard prefixes for 10-based or 2^10-based multipliers) in the numerator and denominator a fraction (for example, "kilobyte/second" or "2.54*centimeter").

Version 3 of this document introduced the following changes in the syntax of programmatic units, compared to version 2.6:

- The set of base units is now extensible by CIM schema implementations (e.g., "acme:myunit").
- SI decimal prefixes can now be used (e.g., "kilobyte").
- IEC binary prefixes can now be used (e.g., "kibibyte").
- Numerical modifiers can now be used multiple times and also as a denominator.
- Floating point numbers can now be used as numerical modifiers (e.g., "2.54*centimeter").
- Integer exponents can now be used on base units (e.g., "meter^2", "second^-2").
- Whitespace between the elements of a complex programmatic unit has been reduced to be only space characters; newline and tab are no longer allowed.
- UCS characters beyond U+007F are no longer allowed in the names of base units.
- "+" as a sign of the exponent of a numerical modifier or as a sign of the entire programmatic unit is no longer allowed in order to remove redundancy.

A base unit of a programmatic unit is a simple unit of measurement with a name and a defined semantic. It is not to be confused with SI base units. The base units of programmatic units can be divided into these groups:

- standard base units; they are defined in Table D-1 extension base units; they can be defined in addition to the standard base units

The name of a standard base unit is a simple identifier string (see the `base-unit` ABNF rule in the syntax below) that is unique within the set of all standard base units listed in Table D-1.

The name of an extension base unit needs to have an additional organization-specific prefix to ensure uniqueness (see the `extension-unit` ABNF rule in the syntax below).

The set of standard base units defined in Table D-1 includes all SI units defined in ISO 1000 and other commonly used units.

The base units of programmatic units can be extended in two ways:

- by adding standard base units in future major or minor versions of this document, or
- by defining extension base units

The string representation of programmatic units is defined by the `programmatic-unit` ABNF rule defined in the syntax below. Any literal strings in this ABNF shall be interpreted case-sensitively and additional whitespace characters shall not be implied to the syntax.

The string representation of programmatic units shall be interpreted using normal mathematical rules. Prefixes bind to the prefixed base unit stronger than an exponent on the prefixed base unit (for example, "millimeter^2" means (0.001*m)^2), consistent with ISO 1000. The comments in the ABNF syntax below describe additional interpretation rules.

```
programmatic-unit  =  [ sign ]  *S  unit-element
                         *( *S  unit-operator  *S  unit-element )
sign  =  HYPHEN
unit-element  =  number  /  [ prefix ]  base-unit  [ CARET  exponent ]
unit-operator  =  "*" / "/"
number  =  floatingpoint-number  /  exponent-number

; An exponent shall be interpreted as a floating point number
; with the specified decimal base and exponent and a mantissa of 1
exponent-number  =  base  CARET  exponent
base  =  integer-number
exponent  =  [ sign ]  integer-number

; An integer shall be interpreted as a decimal integer number
integer-number  =  NON-ZERO-DIGIT  *( DIGIT )

; A float shall be interpreted as a decimal floating point number
floatingpoint-number  =  1*( DIGIT )  [ "." ]  *( DIGIT )

; A prefix for a base unit (e.g. "kilo"). The numeric equivalents of
; these prefixes shall be interpreted as multiplication factors for the
; directly succeeding base unit. In other words, if a prefixed base
; unit is in the denominator of the overall programmatic unit, the
; numeric equivalent of that prefix is also in the denominator
prefix  =  decimal-prefix  /  binary-prefix

; SI decimal prefixes as defined in ISO 1000
decimal-prefix  =
                   "deca"          ; 10^1
                 / "hecto"         ; 10^2
                 / "kilo"          ; 10^3
                 / "mega"          ; 10^6
                 / "giga"          ; 10^9
                 / "tera"          ; 10^12
                 / "peta"          ; 10^15
                 / "exa"           ; 10^18
                 / "zetta"         ; 10^21
                 / "yotta"         ; 10^24
                 / "deci"          ; 10^-1
                 / "centi"         ; 10^-2
                 / "milli"         ; 10^-3
                 / "micro"         ; 10^-6
```

```
                  / "nano"           ; 10^-9
                  / "pico"           ; 10^-12
                  / "femto"          ; 10^-15
                  / "atto"           ; 10^-18
                  / "zepto"          ; 10^-21
                  / "yocto"          ; 10^-24

; IEC binary prefixes as defined in IEC 80000-13
binary-prefix  =
                    "kibi"           ; 2^10
                  / "mebi"           ; 2^20
                  / "gibi"           ; 2^30
                  / "tebi"           ; 2^40
                  / "pebi"           ; 2^50
                  / "exbi"           ; 2^60
                  / "zebi"           ; 2^70
                  / "yobi"           ; 2^80

; The name of a base unit
base-unit  =  standard-unit  /  extension-unit

; The name of a standard base unit
standard-unit  =  UNIT-IDENTIFIER


; The name of an extension base unit. If UNIT-IDENTIFIER begins with a
prefix (see prefix ABNF rule), the meaning of that prefix shall not be
changed by the extension base unit (examples of this for standard base
units are "decibel" or "kilogram")
extension-unit  =  org-id  COLON  UNIT-IDENTIFIER

; org-id shall include a copyrighted, trademarked, or otherwise unique
; name that is owned by the business entity that is defining the
; extension unit, or that is a registered ID assigned to the business
; entity by a recognized global authority. org-id shall not begin with
; a prefix (see prefix ABNF rule)
org-id  =  UNIT-IDENTIFIER
UNIT-IDENTIFIER  =  FIRST-UNIT-CHAR  [ *( MID-UNIT-CHAR )
                    LAST-UNIT-CHAR ]
FIRST-UNIT-CHAR  =  UPPERALPHA  /  LOWERALPHA  /  UNDERSCORE
LAST-UNIT-CHAR  =  FIRST-UNIT-CHAR  /  DIGIT  /  PARENS
MID-UNIT-CHAR  =  LAST-UNIT-CHAR  /  HYPHEN  / S

DIGIT  =  ZERO  /  NON-ZERO-DIGIT
ZERO  =  "0"
NON-ZERO-DIGIT  =  "1"-"9"
HYPHEN  =  U+002D                  ; "-"
CARET  =  U+005E                   ; "^"
COLON  =  U+003A                   ; ":"
UPPERALPHA  =  U+0041-005A     ; "A" ... "Z"
LOWERALPHA  =  U+0061-007A     ; "a" ... "z"
UNDERSCORE  =  U+005F              ; "_"
PARENS  =  U+0028 / U+0029        ; "(", ")"
S  =  U+0020                ; " "
```

For example, a speedometer could be modeled so that the unit of measurement is kilometers per hour. Taking advantage of the SI prefix "kilo" and the fact that "hour" is a standard base unit and thus does not need to be converted to seconds, this unit of measurement can be expressed as a programmatic unit string "kilometer/hour". An alternative way of expressing this programmatic unit string using only SI base units would be "meter/second/3.6".

Other examples are:

"meter*meter*10^-6" → square millimeters
"millimeter*millimeter" → square millimeters
"millimeter^2" → square millimeters
"byte*2^10" → binary kBytes
"1024*byte" → binary kBytes
"kibibyte" → binary kBytes
"byte*10^3" → decimal kBytes
"kilobyte" → decimal kBytes
"dataword*4" → QuadWords
"-decibel-m" → -dBm
"second*250*10^-9" → 250 nanoseconds
"250*nanosecond" → 250 nanoseconds
"foot*foot*foot/minute" → cubic feet per minute, CFM
"foot^3/minute" → cubic feet per minute, CFM
"revolution/minute" → revolutions per minute, RPM
"pound/inch/inch" → pounds per square inch, PSI
"pound/inch^2" → pounds per square inch, PSI
"foot*pound" → foot-pounds

In the "Standard Base Unit" column Table D-1 defines the names of the standard base units. The "Symbol" column recommends a symbol to be used in a human interface. The "Calculation" column relates units to other units. The "Quantity" column lists the physical quantity or quantities measured by the unit.

The standard base units in Table D-1 consist of the SI base units and the SI derived units amended by other commonly used units. "SI" is the international abbreviation for the International System of Units (French: "Système International d'Unites"), defined in ISO 1000.

Table D-1 – Standard base units for programmatic units

Standard Base Unit	Symbol	Calculation	Quantity
			No unit, dimensionless unit (the empty string)
percent	%	1 % = 1/100	Ratio (dimensionless unit)
permille	‰	1 ‰ = 1/1000	Ratio (dimensionless unit)
decibel	dB	1 dB = 10 · lg (P/P0) 1 dB = 20 · lg (U/U0)	Logarithmic ratio (dimensionless unit) Used with a factor of 10 for power, intensity, and so on. Used with a factor of 20 for voltage, pressure, loudness of sound, and so on

Standard Base Unit	Symbol	Calculation	Quantity
Count			Unit for counted items or phenomenons The description of the schema element using this unit should describe what kind of item or phenomenon is counted.
revolution	rev	1 rev = 360°	Turn, plane angle
degree	°	180° = pi rad	Plane angle
Radian	rad	1 rad = 1 m/m	Plane angle
steradian	sr	1 sr = 1 m^2/m^2	Solid angle
Bit	bit		Quantity of information
Byte	B	1 B = 8 bit	Quantity of information
dataword	word	1 word = N bit	Quantity of information The number of bits depends on the computer architecture.
Meter	m	SI base unit	Length (The corresponding ISO SI unit is "metre.")
Inch	in	1 in = 0.0254 m	Length
rack-unit	U	1 U = 1.75 in	Length (height unit used for computer components, as defined in EIA-310)
Foot	ft	1 ft = 12 in	Length
Yard	yd	1 yd = 3 ft	Length
Mile	mi	1 mi = 1760 yd	Length (U.S. land mile)
Liter	l	1000 l = 1 m^3	Volume (The corresponding ISO SI unit is "litre.")
fluid-ounce	fl.oz	33.8140227 fl.oz = 1 l	Volume for liquids (U.S. fluid ounce)
liquid-gallon	gal	1 gal = 128 fl.oz	Volume for liquids (U.S. liquid gallon)
Mole	mol	SI base unit	Amount of substance
kilogram	kg	SI base unit	Mass
Ounce	oz	35.27396195 oz = 1 kg	Mass (U.S. ounce, avoirdupois ounce)
pound	lb	1 lb = 16 oz	Mass (U.S. pound, avoirdupois pound)

Standard Base Unit	Symbol	Calculation	Quantity
second	s	SI base unit	Time (interval)
minute	min	1 min = 60 s	Time (interval)
Hour	h	1 h = 60 min	Time (interval)
Day	d	1 d = 24 h	Time (interval)
Week	week	1 week = 7 d	Time (interval)
Hertz	Hz	1 Hz = 1 /s	Frequency
gravity	g	1 g = 9.80665 m/s²	Acceleration
degree-celsius	°C	1 °C = 1 K (diff)	Thermodynamic temperature
degree-fahrenheit	°F	1 °F = 5/9 K (diff)	Thermodynamic temperature
Kelvin	K	SI base unit	Thermodynamic temperature, color temperature
candela	cd	SI base unit	Luminous intensity
Lumen	lm	1 lm = 1 cd·sr	Luminous flux
Nit	nit	1 nit = 1 cd/m²	Luminance
Lux	lx	1 lx = 1 lm/m²	Illuminance
newton	N	1 N = 1 kg·m/s²	Force
pascal	Pa	1 Pa = 1 N/m²	Pressure
Bar	bar	1 bar = 100000 Pa	Pressure
decibel-A	dB(A)	1 dB(A) = 20 · lg (p/p0)	Loudness of sound, relative to reference sound pressure level of p0 = 20 µPa in gases, using frequency weight curve (A)
decibel-C	dB(C)	1 dB(C) = 20 · lg (p/p0)	Loudness of sound, relative to reference sound pressure level of p0 = 20 µPa in gases, using frequency weight curve (C)
Joule	J	1 J = 1 N·m	Energy, work, torque, quantity of heat
Watt	W	1 W = 1 J/s = 1 V · A	Power, radiant flux In electric power technology, the real power (also known as active power or effective power or true power)
volt-ampere	VA	1 VA = 1 V·A	In electric power technology, the apparent power

Standard Base Unit	Symbol	Calculation	Quantity
volt-ampere-reactive	var	1 var = 1 V·A	In electric power technology, the reactive power (also known as imaginary power)
decibel-m	dBm	1 dBm = 10 · lg (P/P0)	Power, relative to reference power of P0 = 1 mW
british-thermal-unit	BTU	1 BTU = 1055.056 J	Energy, quantity of heat The ISO definition of BTU is used here, out of multiple definitions.
ampere	A	SI base unit	Electric current, magnetomotive force
coulomb	C	1 C = 1 A·s	Electric charge
Volt	V	1 V = 1 W/A	Electric tension, electric potential, electromotive force
Farad	F	1 F = 1 C/V	Capacitance
Ohm	Ohm, Ω	1 Ohm = 1 V/A	Electric resistance
siemens	S	1 S = 1 /Ohm	Electric conductance
weber	Wb	1 Wb = 1 V·s	Magnetic flux
Tesla	T	1 T = 1 Wb/m^2	Magnetic flux density, magnetic induction
Henry	H	1 H = 1 Wb/A	Inductance
becquerel	Bq	1 Bq = 1 /s	Activity (of a radionuclide)
Gray	Gy	1 Gy = 1 J/kg	Absorbed dose, specific energy imparted, kerma, absorbed dose index
sievert	Sv	1 Sv = 1 J/kg	Dose equivalent, dose equivalent index

ANNEX E
(normative)
Operations on timestamps and intervals

E.1 Datetime operations

The following operations are defined on datetime types:

- Arithmetic operations:
  - Adding or subtracting an interval to or from an interval results in an interval.
  - Adding or subtracting an interval to or from a timestamp results in a timestamp.
  - Subtracting a timestamp from a timestamp results in an interval.
  - Multiplying an interval by a numeric or vice versa results in an interval.
  - Dividing an interval by a numeric results in an interval.

  Other arithmetic operations are not defined.
- Comparison operations:
  - Testing for equality of two timestamps or two intervals results in a boolean value.
  - Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in a boolean value.

  Other comparison operations are not defined.

  Comparison between a timestamp and an interval and vice versa is not defined.

Specifications that use the definition of these operations (such as specifications for query languages) should state how undefined operations are handled.

Any operations on datetime types in an expression shall be handled as if the following sequential steps were performed:

1) Each datetime value is converted into a range of microsecond values, as follows:
   - The lower bound of the range is calculated from the datetime value, with any asterisks replaced by their minimum value.
   - The upper bound of the range is calculated from the datetime value, with any asterisks replaced by their maximum value.
   - The basis value for timestamps is the oldest valid value (that is, 0 microseconds corresponds to 00:00.000000 in the time zone with datetime offset +720, on January 1 in the year 1 BCE, using the proleptic Gregorian calendar). This definition implicitly performs timestamp normalization.

NOTE 1 BCE is the year before 1 CE.

2) The expression is evaluated using the following rules for any datetime ranges:
   - Definitions:

     T(x, y) The microsecond range for a timestamp with the lower bound x and the upper bound y

     I(x, y) The microsecond range for an interval with the lower bound x and the upper bound y

D(x, y) The microsecond range for a datetime (timestamp or interval) with the lower bound x and the upper bound y

- Rules:

  I(a, b) + I(c, d) := I(a+c, b+d)
  I(a, b) - I(c, d) := I(a-d, b-c)
  T(a, b) + I(c, d) := T(a+c, b+d)
  T(a, b) - I(c, d) := T(a-d, b-c)
  T(a, b) - T(c, d) := I(a-d, b-c)
  I(a, b) * c := I(a*c, b*c)
  I(a, b) / c := I(a/c, b/c)

  D(a, b) < D(c, d) := true if b < c, false if a >= d, otherwise Null (uncertain)
  D(a, b) <= D(c, d) := true if b <= c, false if a > d, otherwise Null (uncertain)
  D(a, b) > D(c, d) := true if a > d, false if b <= c, otherwise Null (uncertain)
  D(a, b) >= D(c, d) := true if a >= d, false if b < c, otherwise Null (uncertain)
  D(a, b) = D(c, d) := true if a = b = c = d, false if b < c OR a > d, otherwise Null (uncertain)
  D(a, b) <> D(c, d) := true if b < c OR a > d, false if a = b = c = d, otherwise Null (uncertain)

  These rules follow the well-known mathematical interval arithmetic. For a definition of mathematical interval arithmetic, see http://en.wikipedia.org/wiki/Interval_arithmetic.

NOTE 1 Mathematical interval arithmetic is commutative and associative for addition and multiplication, as in ordinary arithmetic.

NOTE 2 Mathematical interval arithmetic mandates the use of three-state logic for the result of comparison operations. A special value called "uncertain" indicates that a decision cannot be made. The special value of "uncertain" is mapped to the Null value in datetime comparison operations.

3) Overflow and underflow condition checking for datetime values is performed on the result of the expression, as follows:

   For timestamp results:

   - A timestamp older than the oldest valid value in the time zone of the result produces an arithmetic underflow condition.
   - A timestamp newer than the newest valid value in the time zone of the result produces an arithmetic overflow condition.

   For interval results:

   - A negative interval produces an arithmetic underflow condition.
   - A positive interval greater than the largest valid value produces an arithmetic overflow condition.

   Specifications using these operations (for example, query languages) should define how these conditions are handled.

4) If the result of the expression is a datetime type, the microsecond range is converted into a valid datetime value such that the set of asterisks (if any) determines a range that matches the actual result range or encloses it as closely as possible. The GMT time zone shall be used for any timestamp results.

NOTE For most fields, asterisks can be used only with the granularity of the entire field.

Examples:

Datetime Expression	Result
"000000000011**.******:000" * 60	"0000000011****.******:000"
60 times adding up "000000000011**.******:000"	"0000000011****.******:000"
"20051003110000.******+000" + "00000000005959.******:000"	"20051003******.******+000"
"20051003112233.******+000" - "00000000002233.******:000"	"20051003******.******+000"
"20051003110000.******+000" + "000000000022**.******:000"	"2005100311****.******+000"
"20051003112233.******+000" - "00000000002232.******:000"	"200510031100**.******+000"
"20051003112233.******+000" - "00000000002233.00000*:000"	"20051003110000.******+000"
"20051003112233.******+000" - "00000000002233.000000:000"	"20051003110000.******+000"
"20051003112233.000000+000" - "00000000002233.000000:000"	"20051003110000.000000+000"
"20051003110000.******+000" + "00000000002233.******:000"	"200510031122**.******+000"
"20051003110000.******+000" + "00000000002233.00000*:000"	"200510031122**.******+000"
"20051003060000.******-300" + "00000000002233.000000:000"	"20051003112233.******+000"
"20051003110000.******+000" + "00000000002233.000000:000"	"20051003112233.******+000"
"20051003060000.000000-300" + "00000000002233.000000:000"	"20051003112233.000000+000"
"20051003110000.000000+000" + "00000000002233.000000:000"	"20051003112233.000000+000"
"20051003112233.******+000" = "200510031122**.******+000"	Null(uncertain)
"20051003112233.******+000" = "20051003112233.******+000"	Null(uncertain)
"20051003112233.5*****+000" < "20051003112233.******+000"	Null(uncertain)
"20051003112233.******+000" = "20051003112234.******+000"	FALSE
"20051003112233.******+000" < "20051003112234.******+000"	TRUE
"20051003112233.000000+000" = "20051003112233.000000+000"	TRUE
"20051003122233.000000+060" = "20051003112233.000000+000"	TRUE

ANNEX F
(normative)
MappingStrings formats

F.1 Mapping entities of other information models to CIM

The MappingStrings qualifierType can be used to map entities of other information models to CIM or to express that a CIM element represents an entity of another information model. Several mapping string formats are defined in this clause to use as values for this qualifierType. The CIM schema shall use only the mapping string formats defined in this document. Extension schemas should use only the mapping string formats defined in this document.

The mapping string formats defined in this document conform to the following formal syntax defined in ABNF:

```
mappingstrings_format = mib_format / oid_format / general_format / mif_format
```

NOTE As defined in the respective clauses, the "MIB", "OID", and "MIF" formats support a limited form of extensibility by allowing an open set of defining bodies. However, the syntax defined for these formats does not allow variations by defining body; they need to conform. A larger degree of extensibility is supported in the general format, where defining bodies might define a part of the syntax used in the mapping.

F.2 SNMP-related mapping string formats

The two SNMP-related mapping string formats, Management Information Base (MIB) and globally unique object identifier (OID), can express that a CIM element represents a MIB variable. As defined in RFC1155, a MIB variable has an associated variable name that is unique within a MIB and an OID that is unique within a management protocol.

The "MIB" mapping string format identifies a MIB variable using naming authority, MIB name, and variable name. The "MIB" mapping string format may be used only on CIM properties, parameters, or methods. The format is defined as follows, using ABNF:

```
mib_format = "MIB" "." mib_naming_authority "|" mib_name "." mib_variable_name
```

Where:

```
mib_naming_authority = 1*(stringChar)
```

is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical bar (|) characters are not allowed.

```
mib_name = 1*(stringChar)
```

is the name of the MIB as defined by the MIB naming authority (for example, "HOST-RESOURCES-MIB"). The dot (.) and vertical bar (|) characters are not allowed.

```
mib_variable_name = 1*(stringChar)
```

is the name of the MIB variable as defined in the MIB (for example, "hrSystemDate"). The dot (.) and vertical bar (|) characters are not allowed.

The MIB name should be the ASN.1 module name of the MIB (that is, not the RFC number). For example, instead of using "RFC1493", the string "BRIDGE-MIB" would be used.
EXAMPLE:

```
   [MappingStrings { "MIB.IETF|HOST-RESOURCES-MIB.hrSystemDate" }]
datetime LocalDateTime;
```

The "OID" mapping string format identifies a MIB variable using a management protocol and an object identifier (OID) within the qualifiedElement of that protocol. This format is especially important for mapping variables defined in private MIBs. The "OID" mapping string format may be used only on CIM properties, parameters, or methods. The format is defined as follows, using ABNF:

```
oid_format = "OID" "." oid_naming_authority "|" oid_protocol_name "." oid
```

Where:

```
oid_naming_authority = 1*(stringChar)
```

is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical bar (|) characters are not allowed.

```
oid_protocol_name = 1*(stringChar)
```

is the name of the protocol providing the qualifiedElement for the OID of the MIB variable (for example, "SNMP"). The dot (.) and vertical bar (|) characters are not allowed.

```
oid = 1*(stringChar)
```

is the object identifier (OID) of the MIB variable in the qualifiedElement of the protocol (for example, "1.3.6.1.2.1.25.1.2").

EXAMPLE:

```
   [MappingStrings { "OID.IETF|SNMP.1.3.6.1.2.1.25.1.2" }]
datetime LocalDateTime;
```

For both mapping string formats, the name of the naming authority defining the MIB shall be one of the following:

- The name of a standards body (for example, IETF), for standard MIBs defined by that standards body
- A company name (for example, Acme), for private MIBs defined by that company

F.3 General mapping string format

This clause defines the mapping string format, which provides a basis for future mapping string formats. A mapping string format based on this format shall define the kinds of CIM elements with which it is to be used.

The format is defined as follows, using ABNF. The division between the name of the format and the actual mapping is slightly different than for the "MIF", "MIB", and "OID" formats:

```
general_format = general_format_fullname "|" general_format_mapping
```

Where:

```
general_format_fullname = general_format_name "." general_format_defining_body

general_format_name = 1*(stringChar)
```

is the name of the format, unique within the defining body. The dot (.) and vertical bar (|) characters are not allowed.

```
general_format_defining_body = 1*(stringChar)
```

is the name of the defining body. The dot (.) and vertical bar (|) characters are not allowed.

```
general_format_mapping = 1*(stringChar)
```

is the mapping of the qualified CIM element, using the named format.

The text in Table F-1 is an example that defines a mapping string format based on the general mapping string format.

Table F-1 – Example MappingStrings mapping

General Mapping String Formats Defined for InfiniBand Trade Association (IBTA)

IBTA defines the following mapping string formats, which are based on the general mapping string format:
 	
`"MAD.IBTA"`
This format expresses that a CIM element represents an IBTA MAD attribute. It shall be used only on CIM properties, parameters, or methods. It is based on the general mapping string format as follows, using ABNF:
 	
`general_format_fullname = "MAD" "." "IBTA"`
 	
`general_format_mapping = mad_class_name "|" mad_attribute_name`
Where:
 	
`mad_class_name = 1*(stringChar)`
is the name of the MAD class. The dot (.) and vertical bar (|) characters are not allowed.
 	
`mad_attribute_name = 1*(stringChar)`
is the name of the MAD attribute, which is unique within the MAD class. The dot (.) and vertical bar (|) characters are not allowed.

ANNEX G
(informative)
Constraint index

ANNEX H
(informative)
Changes from CIM Version 2

- New Features
  - Enumerations (both global and local)
  - Structures (both global and local)
  - Method Overloading - Default value of parameters
  - Method Return Values can be arrays
  - REF in Class
  - All REF props in an association instance must be non-Null
  - REF props are not required to be keys
- No longer Supported
  - Covered Properties

    NOTE covered properties occur when a class and its superclass define properties with the same name but without overriding. The term is an unofficial term that refers to the property of the superclass that is therefore "covered" by the property of the same name in the subclass. CIM v2 deprecated support for covered properties within the same schema. CIM v2 allowed covered properties between a superclass and subclass belonging to different schemas. CIM v3 disallows covered properties in all cases. In the event that a superclass adds properties that conflict with properties of existing subclasses, it is the responsibility of the vendor owning the subclass to resolve the conflict.
  - The ability to use UNICODE Characters within identifiers for schema element names has been removed. The CIM v3 character set for identifiers is specified in A.1.
  - Meta Qualifiers – The Association and Indication qualifiers are no longer supported. CIM v3 covers this functionality
    - CIM v2 classes that have the Indication qualifier can typically be changed to CIM v3 structures. There is no need to further qualify the structure.
    - CIM v2 classes that have the Association qualifier must be changed to CIM v3 associations. The Composition and Aggregation qualifiers are removed from the CIM v3 association. The Aggregate qualifier is removed from the reference to the aggregating class and the AggregationKind is added to the reference to the aggregated class to indicate that instances may be a shared or composed with the aggregating instance
  - The ability to specify a fixed size array using a value within the array brackets has been removed. This functionality is covered in CIM v3 by the use of an OCL qualifier that specifies that the size() of the property or parameter must be a specific value. For properties this is specified as an OCL invariant expression. For parameters the OCL constraint is specified as pre and post condition expressions.
  - The Translatable flavor and therefore the ability to specify language specific qualifier values has been removed
  - Char16 datatype
- New Data Types
  - By reference use of class in structure and class declarations

  - By value use of class
  - By value use of enumeration
  - By value use of structure
  - OctetString
- QualifierType
  - Behavior of flavor vs propagation policy has changed
- Qualifiers
  - New
    - AggregationKind – replaces 3 (Aggregation, Aggregate, Composite)
    - OCL
    - PackagePath replaces UMLPackagePath
  - Modified
    - Override qualifier changed to Boolean
    - Static no longer supports property (continues to support method)
    - ArrayType
      - Set and OrderedSet are added. Both assert that duplicates are not allowed.
- Removed (see Table H-1)

Table H-1: Removed qualifiers

Qualifier	Replaced By	Comments
Aggregate	AggregationKind qualifier	AggregationKind.shared
Alias	No replacement	
Association	Association type	
ClassConstraint	OCL qualifier	Invariant or definition constraint
Composition	AggregationKind qualifier	AggregationKind.composite
Correlatable	No replacement	No replacement
Delete	OCL qualifier	invariant constraint
DisplayDescription	No replacement	No replacement
DisplayName	No replacement	No replacement
DN	No replacement	No replacement
EmbeddedInstance	By value type	
Exception	Structure type	Exception inferred by context
Expensive	No replacement	No replacement
IfDeleted	OCL qualifier	invariant constraint
Indication	Structure type	Indication inferred by context
Invisible	No replacement	No replacement
Large	No replacement	No replacement
MaxLen	OCL qualifier	Example: self.element.size <= MaxLen (see Note 1)
MaxValue	OCL qualifier	Example: self.element <= MaxValue (see Note 1)
MethodConstraint	OCL qualifier	pre/post/body constraint
MinLen	OCL qualifier	Example: self.element.size >= MaxLen (see Note 1)
MinValue	OCL qualifier	Example: self.element >= MaxValue (see Note 1)
NullValue	No replacement	No replacement
OctetString	OctetString type	The length is not part of the representation for values of the OctetString type. Note that this is different from the previous CIM v2 OctetString Qualifier.
Propagated	OCL qualifier	derivation constraint
PropertyConstraint	OCL qualifier	invariant or derivation constraint
PropertyUsage	No replacement	No replacement
Provider	No replacement	No replacement
Reference	Reference type	
Schema	No replacement	No replacement
Structure	Structure type	
Syntax	No replacement	No replacement
SyntaxType	No replacement	No replacement
TriggerType	No replacement	No replacement
UMLPackagePath	PackagePath qualifier	
Units	Punit qualifier	

Qualifier	Replaced By	Comments
UnknownValues	No replacement	No replacement
UnsupportedValues	No replacement	No replacement
ValueMap	Enumeration type	Reserved ranges are not handled by enumeration (see Note 2)
Values	Enumeration type	Reserved ranges are not handled by enumeration (see Note 2)
Weak	OCL qualifier	derivation constraint

NOTE 1 element refers to a property or parameter name, or may be "return" to specify a method return.

NOTE 2 Reserved ranges for string enumerations can be handled by requiring that each enumeration be prefixed with an organization specific prefix (e.g., Golf_). Reserved ranges for string or integer enumerations can be handled by adding a separate, schema specific enumeration and then using that enumeration as a separate property or parameter. The use of additional enumerations can be in addition or an extension to an existing enumeration. If in addition, the added enumerations need to make sense in the context of the existing enumerations. OCL qualifiers can be used to restrict combinations. If used as an extension, the original enumeration would be extended to indicate that extension schema specific values are used instead of those of the extended schema.

ANNEX I
(informative)
Change log

Version	Date	Description
1.0.0	1997-04-09	
2.2.0	1999-06-14	Released as Final Standard
2.2.1000	2003-06-07	Released as Final Standard
2.3.0	2005-10-04	Released as Final Standard
2.5.0	2009-03-04	Released as DMTF Standard
2.6.0	2010-03-17	Released as DMTF Standard
2.7.0	2012-04-22	Released as DMTF Standard
3.0.0	2012-12-13	Released as DMTF Standard

Bibliography

DMTF DSP0200, CIM operations over HTTP, Version 1.3
http://www.dmtf.org/standards/published_documents/DSP0200_1.3.pdf

IEEE Std 1003.1, 2004 Edition, Standard for information technology - portable operating system interface (POSIX). Shell and utilities
http://www.unix.org/version3/ieee_std.html

IETF, RFC1155, Structure and Identification of Management Information for TCP/IP-based Internets, http://tools.ietf.org/html/rfc1155

ISO/IEC 14651:2007, Information technology — International string ordering and comparison — Method for comparing character strings and description of the common template tailorable ordering
http://standards.iso.org/ittf/PubliclyAvailableStandards/c044872_ISO_IEC_14651_2007(E).zip

ISO/IEC 19757-2:2008, Information technology -- Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG,
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52348

OMG MOF 2 XMI Mapping Specification, formal/2011-08-09, version 2.4.1
http://www.omg.org/spec/XMI/2.4.1

The Unicode Consortium. The Unicode Standard, Version 6.1.0, (Mountain View, CA: The Unicode Consortium, 2012. ISBN 978-1-936213-02-3)
http://www.unicode.org/versions/Unicode6.1.0/

W3C, XML Schema Part 0: Primer Second Edition, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-0/