**Document Number: DSP0004**

**Date: 2012-04-22**

**Version: 2.7.0**

# Common Information Model (CIM) Infrastructure

**Document Type: Specification**

**Document Status: DMTF Standard**

**Document Language: en-US**

## Trademarks

- Microsoft is a registered trademark of Microsoft Corporation.
- UNIX is registered trademark of The Open Group.

# CONTENTS

## Figures



## Tables

# Foreword

The *Common Information Model (CIM) Infrastructure* (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:

- Lawrence Lamers – VMware

Contributors:

- Jeff Piazza – Hewlett-Packard Company
- Andreas Maier – IBM
- George Ericson – EMC
- Jim Davis – WBEM Solutions
- Karl Schopmeyer – Inova Development
- Steve Hand – Symantec
- Andrea Westerinen – CA Technologies
- Aaron Merkin - Dell

# Introduction

The Common Information Model (CIM) can be used in many ways. Ideally, information for performing tasks is organized so that disparate groups of people can use it. This can be accomplished through an information model that represents the details required by people working within a particular domain. An information model requires a set of legal statement types or syntax to capture the representation and a collection of expressions to manage common aspects of the domain (in this case, complex computer systems). Because of the focus on common aspects, the Distributed Management Task Force (DMTF) refers to this information model as CIM, the Common Information Model. For information on the current core and common schemas developed using this meta model, contact the DMTF.

## 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, OCL text and CIM MOF 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.
- 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)
- 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.

### Deprecated Material

Deprecated material is not recommended for use in new development efforts. Existing and new implementations may use this material, but they shall move to the favored approach as soon as possible. CIM servers shall implement any deprecated elements as required by this document in order to achieve backwards compatibility. Although CIM clients may use deprecated elements, they are directed to use the favored elements instead.

Deprecated material should contain references to the last published version that included the deprecated material as normative material and to a description of the favored approach.

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 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.

## CIM Management Schema

Management schemas are the building-blocks for management platforms and management applications, such as device configuration, performance management, and change management. CIM structures the managed environment as a collection of interrelated systems, each composed of discrete elements.

CIM supplies a set of classes with properties and associations that provide a well-understood conceptual framework to organize the information about the managed environment. We assume a thorough knowledge of CIM by any programmer writing code to operate against the object schema or by any schema designer intending to put new information into the managed environment.

CIM is structured into these distinct layers: core model, common model, extension schemas.

## Core Model

The core model is an information model that applies to all areas of management. The core model is a small set of classes, associations, and properties for analyzing and describing managed systems. It is a starting point for analyzing how to extend the common schema. While classes can be added to the core model over time, major reinterpretations of the core model classes are not anticipated.

## Common Model

The common model is a basic set of classes that define various technology-independent areas, such as systems, applications, networks, and devices. The classes, properties, associations, and methods in the common model are detailed enough to use as a basis for program design and, in some cases, implementation. Extensions are added below the common model in platform-specific additions that supply concrete classes and implementations of the common model classes. As the common model is extended, it offers a broader range of information.

The common model is an information model common to particular management areas but independent of a particular technology or implementation. The common areas are systems, applications, networks, and devices. The information model is specific enough to provide a basis for developing management applications. This schema provides a set of base classes for extension into the area of technology-specific schemas. The core and common models together are referred to in this document as the CIM schema.

## Extension Schema

The extension schemas are technology-specific extensions to the common model. Operating systems (such as Microsoft Windows® or UNIX®) are examples of extension schemas. The common model is expected to evolve as objects are promoted and properties are defined in the extension schemas.

## CIM Implementations

Because CIM is not bound to a particular implementation, it can be used to exchange management information in a variety of ways; four of these ways are illustrated in Figure 1. These ways of exchanging information can be used in combination within a management application.

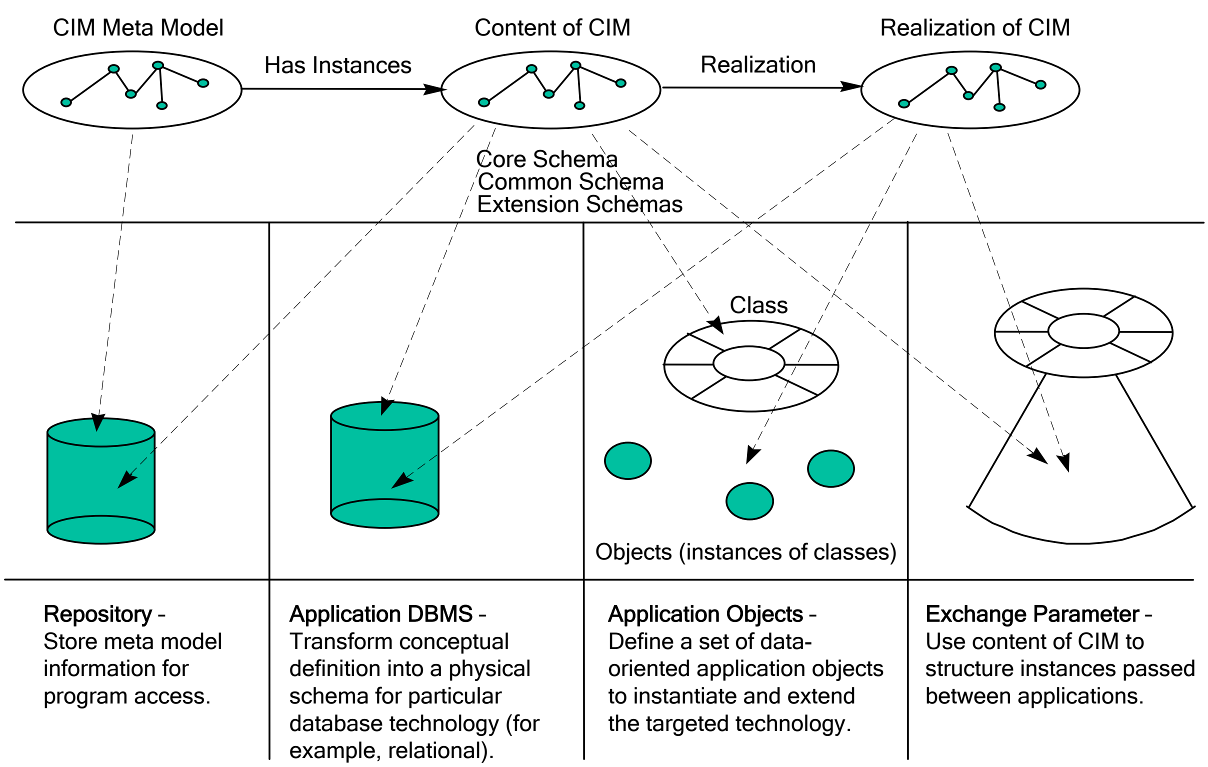



**Figure 1 – Four Ways to Use CIM**

The constructs defined in the model are stored in a database repository. These constructs are not instances of the object, relationship, and so on. Rather, they are definitions to establish objects and relationships. The meta model used by CIM is stored in a repository that becomes a representation of the meta model. The constructs of the meta-model are mapped into the physical schema of the targeted repository. Then the repository is populated with the classes and properties expressed in the core model, common model, and extension schemas.

For an application database management system (DBMS), the CIM is mapped into the physical schema of a targeted DBMS (for example, relational). The information stored in the database consists of actual instances of the constructs. Applications can exchange information when they have access to a common DBMS and the mapping is predictable.

For application objects, the CIM is used to create a set of application objects in a particular language. Applications can exchange information when they can bind to the application objects.

For exchange parameters, the CIM — expressed in some agreed syntax — is a neutral form to exchange management information through a standard set of object APIs. The exchange occurs through a direct set of API calls or through exchange-oriented APIs that can create the appropriate object in the local implementation technology.

## CIM Implementation Conformance

An implementation of CIM is conformant to this specification if it satisfies all requirements defined in this specification.

# Common Information Model (CIM) Infrastructure

## 1 Scope

The DMTF Common Information Model (CIM) Infrastructure is an approach to the management of systems and networks that applies the basic structuring and conceptualization techniques of the object-oriented paradigm. The approach uses a uniform modeling formalism that together with the basic repertoire of object-oriented constructs supports the cooperative development of an object-oriented schema across multiple organizations.

This document describes an object-oriented meta model based on the Unified Modeling Language (UML). This model includes expressions for common elements that must be clearly presented to management applications (for example, object classes, properties, methods, and associations).

This document does not describe specific CIM implementations, application programming interfaces (APIs), or communication protocols.

## 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.

Table 1 shows standards bodies and their web sites.

**Table 1 – Standards Bodies**

| Abbreviation | Standards Body | Web Site |
|---|---|---|
| ANSI | American National Standards Institute | http://www.ansi.org |
| DMTF | Distributed Management Task Force | http://www.dmtf.org |
| EIA | Electronic Industries Alliance | http://www.eia.org |
| IEC | International Engineering Consortium | http://www.iec.ch |
| IEEE | Institute of Electrical and Electronics Engineers | http://www.ieee.org |
| IETF | Internet Engineering Task Force | http://www.ietf.org |
| INCITS | International Committee for Information Technology Standards | http://www.incits.org |
| ISO | International Standards Organization | http://www.iso.ch |
| ITU | International Telecommunications Union | http://www.itu.int |
| W3C | World Wide Web Consortium | http://www.w3.org |

ANSI/IEEE 754-1985, *IEEE® Standard for BinaryFloating-Point Arithmetic*, August 1985
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=30711

DMTF DSP0207, *WBEM URI Mapping Specification*, Version 1.0
http://www.dmtf.org/standards/published_documents/DSP0207_1.0.pdf

DMTF DSP4004, *DMTF Release Process*, Version 2.2
http://www.dmtf.org/standards/published_documents/DSP4004_2.2.pdf

EIA-310, *Cabinets, Racks, Panels, and Associated Equipment*
http://electronics.ihs.com/collections/abstracts/eia-310.htm

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 RFC3986, *Uniform Resource Identifiers (URI): Generic Syntax*, August 1998
http://tools.ietf.org/html/rfc2396

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 639-1:2002, *Codes for the representation of names of languages — Part 1: Alpha-2 code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109

ISO 639-2:1998, *Codes for the representation of names of languages — Part 2: Alpha-3 code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=4767

ISO 639-3:2007, *Codes for the representation of names of languages — Part 3: Alpha-3 code for comprehensive coverage of languages*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39534

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 3166-1:2006, *Codes for the representation of names of countries and their subdivisions — Part 1: Country codes*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39719

ISO 3166-2:2007, *Codes for the representation of names of countries and their subdivisions — Part 2: Country subdivision code*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39718

ISO 3166-3:1999, *Codes for the representation of names of countries and their subdivisions — Part 3: Code for formerly used names of countries*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=2130

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

ISO/IEC 9075-10:2003, *Information technology — Database languages — SQL — Part 10: Object Language Bindings (SQL/OLB)*
http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=34137

ISO/IEC 10165-4:1992, *Information technology — Open Systems Interconnection – Structure of management information — Part 4: Guidelines for the definition of managed objects (GDMO)*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=18174

ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003(E).zip

ISO/IEC 10646:2003/Amd 1:2005, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Amendment 1: Glagolitic, Coptic, Georgian and other characters*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip

ISO/IEC 10646:2003/Amd 2:2006, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Amendment 2: N'Ko, Phags-pa, Phoenician and other characters*
http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip

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 14750:1999, *Information technology — Open Distributed Processing — Interface Definition Language*
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25486

ITU X.501, *Information Technology — Open Systems Interconnection — The Directory: Models*
http://www.itu.int/rec/T-REC-X.501/en

ITU X.680 (07/02), *Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*
http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf

OMG, *Object Constraint Language*, Version 2.0
http://www.omg.org/cgi-bin/doc?formal/2006-05-01

OMG, *Unified Modeling Language: Superstructure*, Version 2.1.1
http://www.omg.org/cgi-bin/doc?formal/07-02-05

The Unicode Consortium, *The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms*
http://www.unicode.org/reports/tr15/

W3C, *Namespaces 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**

**address**

the general concept of a location reference to a CIM object that is accessible through a CIM server, not implying any particular format or protocol

More specific kinds of addresses are object paths.

Embedded objects are not addressable; they may be accessible indirectly through their embedding instance. Instances of an indication class are not addressable since they only exist while being delivered.

**3.2**

**aggregation**

a strong form of association that expresses a whole-part relationship between each instance on the aggregating end and the instances on the other ends, where the instances on the other ends can exist independently from the aggregating instance.

For example, the containment relationship between a physical server and its physical components can be considered an aggregation, since the physical components can exist if the server is dismantled. A stronger form of aggregation is a composition.

**3.3**

**ancestor**

the ancestor of a schema element is for a class, its direct superclass (if any); for a property or method, its overridden property or method (if any); and for a parameter of a method, the like-named parameter of the overridden method (if any)

The ancestor of a schema element plays a role for propagating qualifier values to that schema element for qualifiers with flavor ToSubclass.

**3.4**

**ancestry**

the ancestry of a schema element is the set of schema elements that results from recursively determining its ancestor schema elements

A schema element is not considered part of its ancestry.

**3.5**

**arity**

the number of references exposed by an association class

**3.6**

**association, CIM association**

a special kind of class that expresses the relationship between two or more other classes

The relationship is established by two or more references defined in the association that are typed to a class the referenced instances are of.

For example, an association ACME_SystemDevice may relate the classes ACME_System and ACME_Device by defining references to those classes.

A CIM association is a UML association class. Each has the aspects of both a UML association and a UML class, which may expose ordinary properties and methods and may be part of a class inheritance hierarchy. The references belonging to a CIM association belong to it and are also exposed as part of the association and not as parts of the associated classes. The term "association class" is sometimes used instead of the term "association" when the class aspects of the element are being emphasized.

Aggregations and compositions are special kinds of associations.

In a CIM server, associations are special kinds of objects. The term "association object" (i.e., object of association type) is sometimes used to emphasize that. The address of such association objects is termed "class path", since associations are special classes. Similarly, association instances are a special kind of instances and are also addressable objects. Associations may also be represented as embedded instances, in which case they are not independently addressable.

In a schema, associations are special kinds of schema elements.
In the CIM meta-model, associations are represented by the meta-element named "Association".

**3.7**

**association end**

a synonym for the reference defined in an association

**3.8**

**cardinality**

the number of instances in a set

---

**DEPRECATED**

The use of the term "cardinality" for the allowable range for the number of instances on an association end is deprecated. The term "multiplicity" has been introduced for that, consistent with UML terminology.

**DEPRECATED**

---

**3.9**

**Common Information Model**

**CIM**

CIM (Common Information Model) is:

1. the name of the meta-model used to define schemas (e.g., the CIM schema or extension schemas).
2. the name of the schema published by the DMTF (i.e., the CIM schema).

**3.10**

**CIM schema**

the schema published by the DMTF that defines the Common Information Model
It is divided into a core model and a common model. Extension schemas are defined outside of the DMTF and are not considered part of the CIM schema.

**3.11**

**CIM client**

a role responsible for originating CIM operations for processing by a CIM server
This definition does not imply any particular implementation architecture or scope, such as a client library component or an entire management application.

**3.12**

**CIM listener**

a role responsible for processing CIM indications originated by a CIM server
This definition does not imply any particular implementation architecture or scope, such as a standalone demon component or an entire management application.

**3.13**

**CIM operation**

an interaction within a CIM protocol that is originated by a CIM client and processed by a CIM server

**3.14**

**CIM protocol**

a protocol that is used between CIM client, CIM server and CIM listener
This definition does not imply any particular communication protocol stack, or even that the protocol performs a remote communication.

**3.15**

**CIM server**

a role responsible for processing CIM operations originated by a CIM client and for originating CIM indications for processing by a CIM listener

This definition does not imply any particular implementation architecture, such as a separation into a CIMOM and provider components.

**3.16**

**class, CIM class**

a common type for a set of instances that support the same features

A class is defined in a schema and models an aspect of a managed object. For a full definition, see 5.1.2.7.

For example, a class named "ACME_Modem" may represent a common type for instances of modems and may define common features such as a property named "ActualSpeed" to represent the actual modem speed.

Special kinds of classes are ordinary classes, association classes and indication classes.

In a CIM server, classes are special kinds of objects. The term "class object" (i.e., object of class type) is sometimes used to emphasize that. The address of such class objects is termed "class path".

In a schema, classes are special kinds of schema elements.

In the CIM meta-model, classes are represented by the meta-element named "Class".

**3.17**

**class declaration**

the definition (or specification) of a class

For example, a class that is accessible through a CIM server can be retrieved by a CIM client. What the CIM client receives as a result is actually the class declaration. Although unlikely, the class accessible through the CIM server may already have changed its definition by the time the CIM client receives the class declaration. Similarly, when a class accessible through a CIM server is being modified through a CIM operation, one input parameter might be a class declaration that is used during the processing of the CIM operation to change the class.

**3.18**

**class path**

a special kind of object path addressing a class that is accessible through a CIM server

**3.19**

**class origin**

the class origin of a feature is the class defining the feature

**3.20**

**common model**

the subset of the CIM Schema that is specific to particular domains

It is derived from the core model and is actually a collection of models, including (but not limited to) the System model, the Application model, the Network model, and the Device model.

**3.21**

**composition**

a strong form of association that expresses a whole-part relationship between each instance on the aggregating end and the instances on the other ends, where the instances on the other ends cannot exist independently from the aggregating instance

For example, the containment relationship between a running operating system and its logical devices can be considered a composition, since the logical devices cannot exist if the operating system does not exist. A composition is also a strong form of aggregation.

**3.22**

**core model**

the subset of the CIM Schema that is not specific to any particular domain

The core model establishes a basis for derived models such as the common model or extension schemas.

**3.23**

**creation class**

the creation class of an instance is the most derived class of the instance

The creation class of an instance can also be considered the factory of the instance (although in CIM, instances may come into existence through other means than issuing an instance creation operation against the creation class).

**3.24**

**domain**

an area of management or expertise

---

**DEPRECATED**

The following use of the term "domain" is deprecated: The domain of a feature is the class defining the feature. For example, if class ACME_C1 defines property P1, then ACME_C1 is said to be the domain of P1. The domain acts as a space for the names of the schema elements it defines in which these names are unique. Use the terms "class origin" or "class defining the schema element" or "class exposing the schema element" instead.

**DEPRECATED**

---

**3.25**

**effective qualifier value**

For every schema element, an effective qualifier value can be determined for each qualifier scoped to the element. The effective qualifier value on an element is the value that determines the qualifier behavior for the element.

For example, qualifier Counter is defined with flavor ToSubclass and a default value of False. If a value of True is specified for Counter on a property NumErrors in a class ACME_Device, then the effective value of qualifier Counter on that property is True. If an ACME_Modem subclass of class ACME_Device overrides NumErrors without specifying the Counter qualifier again, then the effective value of qualifier Counter on that property is also True since its flavor ToSubclass defines that the effective value of qualifier Counter is determined from the next ancestor element of the element that has the qualifier specified.

**3.26**

**element**

a synonym for schema element

**3.27**

**embedded class**

a class declaration that is embedded in the value of a property, parameter or method return value

**3.28**

**embedded instance**

an instance declaration that is embedded in the value of a property, parameter or method return value

**3.29**

**embedded object**

an embedded class or embedded instance

**3.30**

**explicit qualifier**

a qualifier type declared separately from its usage on schema elements

See also implicit qualifier.

**3.31**

**extension schema**

a schema not owned by the DMTF whose classes are derived from the classes in the CIM Schema

**3.32**

**feature**

a property or method defined in a class

A feature is exposed if it is available to consumers of a class. The set of features exposed by a class is the union of all features defined in the class and its ancestry. In the case where a feature overrides a feature, the combined effects are exposed as a single feature.

**3.33**

**flavor**

meta-data on a qualifier type that defines the rules for propagation, overriding and translatability of qualifiers

For example, the Key qualifier has the flavors ToSubclass and DisableOverride, meaning that the qualifier value gets propagated to subclasses and these subclasses cannot override it.

**3.34**

**implicit qualifier**

a qualifier type declared as part of the declaration of a schema element

See also explicit qualifier.

---

**DEPRECATED**

The concept of implicitly defined qualifier types (i.e., implicit qualifiers) is deprecated. See 5.1.2.16 for details.

**DEPRECATED**

---

**3.35**

**indication, CIM indication**

a special kind of class that expresses the notification about an event that occurred

Indications are raised based on a trigger that defines the condition under which an event causes an indication to be raised. Events may be related to objects accessible in a CIM server, such as the creation, modification, deletion of or access to an object, or execution of a method on the object. Events may also be related to managed objects, such as alerts or errors.

For example, an indication ACME_AlertIndication may express the notification about an alert event.

The term "indication class" is sometimes used instead of the term "indication" to emphasize that an indication is also a class.

In a CIM server, indication instances are not addressable. They exist as embedded instances in the protocol message that delivers the indication.

In a schema, indications are special kinds of schema elements.

In the CIM meta-model, indications are represented by the meta-element named "Indication".

The term "indication" also refers to an interaction within a CIM protocol that is originated on a CIM server and processed by a CIM listener.

**3.36**

**inheritance**

a relationship between a more general class and a more specific class

An instance of the specific class is also an instance of the general class. The specific class inherits the features of the general class. In an inheritance relationship, the specific class is termed "subclass" and the general class is termed "superclass".

For example, if a class ACME_Modem is a subclass of a class ACME_Device, any ACME_Modem instance is also an ACME_Device instance.

**3.37**

**instance, CIM instance**

This term has two (different) meanings:

1) As instance of a class:

   An instance of a class has values (including possible Null) for the properties exposed by its creation class. Embedded instances are also instances.

   In a CIM server, instances are special kinds of objects. The term "instance object" (i.e., object of instance type) is sometimes used to emphasize that. The address of such instance objects is termed "instance path".

   In a schema, instances are special kinds of schema elements.

   In the CIM meta-model, instances are represented by the meta-element named "Instance".

2) As instance of a meta-element:

   A relationship between an element and its meta-element. For example, a class ACME_Modem is said to be an instance of the meta-element Class, and a property ACME_Modem.Speed is said to be an instance of the meta-element Property.

**3.38**

**instance path**

a special kind of object path addressing an instance that is accessible through a CIM server

**3.39**

**instance declaration**

the definition (or specification) of an instance by means of specifying a creation class for the instance and a set of property values

For example, an instance that is accessible through a CIM server can be retrieved by a CIM client. What the CIM client receives as a result, is actually an instance declaration. The instance itself may already have changed its property values by the time the CIM client receives the instance declaration. Similarly, when an instance that is accessible through a CIM server is being modified through a CIM operation, one input parameter might be an instance declaration that specifies the intended new property values for the instance.

**3.40**

**key**

The key of an instance is synonymous with the model path of the instance (class name, plus set of key property name/value pairs). The key of an instance is required to be unique in the namespace in which it is registered. The key properties of a class are indicated by the Key qualifier.

Also, shorthand for the term "key property".

**3.41**

**managed object**

a resource in the managed environment of which an aspect is modeled by a class

An instance of that class represents that aspect of the represented resource.

For example, a network interface card is a managed object whose logical function may be modeled as a class ACME_NetworkPort.

**3.42**

**meta-element**

an entity in a meta-model

The boxes in Figure 2 represent the meta-elements defined in the CIM meta-model.

For example, the CIM meta-model defines a meta-element named "Property" that defines the concept of a structural data item in an object. Specific properties (e.g., property P1) can be thought of as being instances of the meta-element named "Property".

**3.43**

**meta-model**

a set of meta-elements and their meta-relationships that expresses the types of things that can be defined in a schema

For example, the CIM meta-model includes the meta-elements named "Property" and "Class" which have a meta-relationship such that a Class owns zero or more Properties.

**3.44**

**meta-relationship**

a relationship between two entities in a meta-model

The links in Figure 2 represent the meta-relationships defined in the CIM meta-model.

For example, the CIM meta-model defines a meta-relationship by which the meta-element named "Property" is aggregated into the meta-element named "Class".

**3.45**

**meta-schema**

a synonym for meta-model

**3.46**

**method, CIM method**

a behavioral feature of a class

Methods can be invoked to produce the associated behavior.

In a schema, methods are special kinds of schema elements. Method name, return value, parameters and other information about the method are defined in the class declaration.

In the CIM meta-model, methods are represented by the meta-element named "Method".

**3.47**

**model**

a set of classes that model a specific domain

A schema may contain multiple models (that is the case in the CIM Schema), but a particular domain could also be modeled using multiple schemas, in which case a model would consist of multiple schemas.

**3.48**

**model path**

the part of an object path that identifies the object within the namespace

**3.49**

**multiplicity**

The multiplicity of an association end is the allowable range for the number of instances that may be associated to each instance referenced by each of the other ends of the association. The multiplicity is defined on a reference using the Min and Max qualifiers.

**3.50**

**namespace, CIM namespace**

a special kind of object that is accessible through a CIM server that represents a naming space for classes, instances and qualifier types

**3.51**

**namespace path**

a special kind of object path addressing a namespace that is accessible through a CIM server

Also, the part of an instance path, class path and qualifier type path that addresses the namespace.

**3.52**

**name**

an identifier that each element or meta-element has in order to identify it in some scope

---

**DEPRECATED**

The use of the term "name" for the address of an object that is accessible through a CIM server is deprecated. The term "object path" should be used instead.

**DEPRECATED**

---

**3.53**

**object, CIM object**

a class, instance, qualifier type or namespace that is accessible through a CIM server

An object may be addressable, i.e., have an object path. Embedded objects are objects that are not addressable; they are accessible indirectly through their embedding property, parameter or method return value. Instances of indications are objects that are not addressable either, as they are not accessible through a CIM server at all and only exist in the protocol message in which they are being delivered.

---

**DEPRECATED**

The term "object" has historically be used to mean just "class or instance". This use of the term "object" is deprecated. If a restriction of the term "object" to mean just "class or instance" is intended, this is now stated explicitly.

**DEPRECATED**

---

**3.54**

**object path**

the address of an object that is accessible through a CIM server

An object path consists of a namespace path (addressing the namespace) and optionally a model path (identifying the object within the namespace).

**3.55**

**ordinary class**

a class that is neither an association class nor an indication class

**3.56**

**ordinary property**

a property that is not a reference

**3.57**

**override**

a relationship between like-named elements of the same type of meta-element in an inheritance hierarchy, where the overriding element in a subclass redefines the overridden element in a superclass

The purpose of an override relationship is to refine the definition of an element in a subclass.

For example, a class ACME_Device may define a string typed property Status that may have the values "powersave", "on", or "off". A class ACME_Modem, subclass of ACME_Device, may override the Status property to have only the values "on" or "off", but not "powersave".

**3.58**

**parameter, CIM parameter**

a named and typed argument passed in and out of methods

The return value of a method is not considered a parameter; instead it is considered part of the method.

In a schema, parameters are special kinds of schema elements.

In the CIM meta-model, parameters are represented by the meta-element named "Parameter".

**3.59**

**polymorphism**

the ability of an instance to be of a class and all of its subclasses

For example, a CIM operation may enumerate all instances of class ACME_Device. If the instances returned may include instances of subclasses of ACME_Device, then that CIM operation is said to implement polymorphic behavior.

**3.60**

**propagation**

the ability to derive a value of one property from the value of another property

CIM supports propagation via either PropertyConstraint qualifiers utilizing a derivation constraint or via weak associations.

**3.61**

**property, CIM property**

a named and typed structural feature of a class

Name, data type, default value and other information about the property are defined in a class. Properties have values that are available in the instances of a class. The values of its properties may be used to characterize an instance.

For example, a class ACME_Device may define a string typed property named "Status". In an instance of class ACME_Device, the Status property may have a value "on".

Special kinds of properties are ordinary properties and references.

In a schema, properties are special kinds of schema elements.

In the CIM meta-model, properties are represented by the meta-element named "Property".

**3.62**

**qualified element**

a schema element that has a qualifier specified in the declaration of the element

For example, the term "qualified element" in the description of the Counter qualifier refers to any property (or other kind of schema element) that has the Counter qualifier specified on it.

**3.63**

**qualifier, CIM qualifier**

a named value used to characterize schema elements

Qualifier values may change the behavior or semantics of the qualified schema element. Qualifiers can be regarded as metadata that is attached to the schema elements. The scope of a qualifier determines on which kinds of schema elements a specific qualifier can be specified.

For example, if property ACME_Modem.Speed has the Key qualifier specified with a value of True, this characterizes the property as a key property for the class.

**3.64**

**qualifier type**

a common type for a set of qualifiers

In a CIM server, qualifier types are special kinds of objects. The address of qualifier type objects is termed "qualifier type path".

In a schema, qualifier types are special kinds of schema elements.

In the CIM meta-model, qualifier types are represented by the meta-element named "QualifierType".

**3.65**

**qualifier type declaration**

the definition (or specification) of a qualifier type

For example, a qualifier type object that is accessible through a CIM server can be retrieved by a CIM client. What the CIM client receives as a result, is actually a qualifier type declaration. Although unlikely, the qualifier type itself may already have changed its definition by the time the CIM client receives the qualifier type declaration. Similarly, when a qualifier type that is accessible through a CIM server is being modified through a CIM operation, one input parameter might be a qualifier type declaration that is used during the processing of the operation to change the qualifier type.

**3.66**

**qualifier type path**

a special kind of object path addressing a qualifier type that is accessible through a CIM server

**3.67**

**qualifier value**

the value of a qualifier in a general sense, without implying whether it is the specified value, the effective value, or the default value

**3.68**

**reference, CIM reference**

an association end

References are special kinds of properties that reference an instance.

The value of a reference is an instance path. The type of a reference is a class of the referenced instance. The referenced instance may be of a subclass of the class specified as the type of the reference.

In a schema, references are special kinds of schema elements.

In the CIM meta-model, references are represented by the meta-element named "Reference".

**3.69**

**schema**

a set of classes with a single defining authority or owning organization

In the CIM meta-model, schemas are represented by the meta-element named "Schema".

**3.70**

**schema element**

a specific class, property, method or parameter

For example, a class ACME_C1 or a property P1 are schema elements.

**3.71**

**scope**

part of a qualifier type, indicating the meta-elements on which the qualifier can be specified

For example, the Abstract qualifier has scope class, association and indication, meaning that it can be specified only on ordinary classes, association classes, and indication classes.

**3.72**

**scoping object, scoping instance, scoping class**

a scoping object provides context for a set of other objects

A specific example is an object (class or instance) that propagates some or all of its key properties to a weak object, along a weak association.

**3.73**

**signature**

a method name together with the type of its return value and the set of names and types of its parameters

**3.74**

**subclass**

See inheritance.

**3.75**

**superclass**

See inheritance.

**3.76**

**top-level object**

---

**DEPRECATED**

The use of the terms "top-level object" or "TLO" for an object that has no scoping object is deprecated. Use phrases like "an object that has no scoping object", instead.

**DEPRECATED**

---

**3.77**

**trigger**

a condition that when True, expresses the occurrence of an event

**3.78**

**UCS character**

A character from the Universal Multiple-Octet Coded Character Set (UCS) defined in ISO/IEC 10646:2003. For details, see 5.2.1.

**3.79**

**weak object, weak instance, weak class**

an object (class or instance) that gets some or all of its key properties propagated from a scoping object, along a weak association

**3.80**

**weak association**

an association that references a scoping object and weak objects, and along which the values of key properties get propagated from a scoping object to a weak object

In the weak object, the key properties to be propagated have qualifier Propagate with an effective value of True, and the weak association has qualifier Weak with an effective value of True on its end referencing the weak object.

# 4 Symbols and Abbreviated Terms

The following abbreviations are used in this document.

**4.1**

**API**

application programming interface

**4.2**

**CIM**

Common Information Model

**4.3**

**DBMS**

Database Management System

**4.4**

**DMI**

Desktop Management Interface

**4.5**

**GDMO**

Guidelines for the Definition of Managed Objects

**4.6**

**HTTP**

Hypertext Transfer Protocol

**4.7**

**MIB**

Management Information Base

**4.8**

**MIF**

Management Information Format

**4.9**

**MOF**

Managed Object Format

**4.10**

**OID**

object identifier

**4.11**

**SMI**

Structure of Management Information

**4.12**

**SNMP**

Simple Network Management Protocol

**4.13**

**UML**

Unified Modeling Language

# 5 Meta Schema

The Meta Schema is a formal definition of the model that defines the terms to express the model and its usage and semantics (see ANNEX B).

The Unified Modeling Language (UML) (see *Unified Modeling Language: Superstructure*) defines the structure of the meta schema. In the discussion that follows, italicized words refer to objects in Figure 2. We assume familiarity with UML notation (see www.rational.com/uml) and with basic object-oriented concepts in the form of classes, properties, methods, operations, inheritance, associations, objects, cardinality, and polymorphism.

## 5.1 Definition of the Meta Schema

The CIM meta schema provides the basis on which CIM schemas and models are defined. The CIM meta schema defines meta-elements that have attributes and relationships between them. For example, a CIM class is a meta-element that has attributes such as a class name, and relationships such as a generalization relationship to a superclass, or ownership relationships to its properties and methods.

The CIM meta schema is defined as a UML user model, using the following UML concepts:

- CIM meta-elements are represented as UML classes (UML Class metaclass defined in *Unified Modeling Language: Superstructure*)
- CIM meta-elements may use single inheritance, which is represented as UML generalization (UML Generalization metaclass defined in *Unified Modeling Language: Superstructure*)
- Attributes of CIM meta-elements are represented as UML properties (UML Property metaclass defined in *Unified Modeling Language: Superstructure*)
- Relationships between CIM meta-elements are represented as UML associations (UML Association metaclass defined in *Unified Modeling Language: Superstructure*) whose association ends are owned by the associated metaclasses. The reason for that ownership is that UML Association metaclasses do not have the ability to own attributes or operations. Such relationships are defined in the "Association ends" sections of each meta-element definition.

Languages defining CIM schemas and models (e.g., CIM Managed Object Format) shall use the meta-schema defined in this subclause, or an equivalent meta-schema, as a basis.

A meta schema describing the actual run-time objects in a CIM server is not in scope of this CIM meta schema. Such a meta schema may be closely related to the CIM meta schema defined in this subclause, but there are also some differences. For example, a CIM instance specified in a schema or model following this CIM meta schema may specify property values for a subset of the properties its defining class exposes, while a CIM instance in a CIM server always has all properties exposed by its defining class.

Any statement made in this document about a kind of CIM element also applies to sub-types of the element. For example, any statement made about classes also applies to indications and associations. In some cases, for additional clarity, the sub-types to which a statement applies, is also indicated in parenthesis (example: "classes (including association and indications)").

If a statement is intended to apply only to a particular type but not to its sub-types, then the additional qualification "ordinary" is used. For example, an ordinary class is a class that is not an indication or an association.

Figure 2 shows a UML class diagram with all meta-elements and their relationships defined in the CIM meta schema.

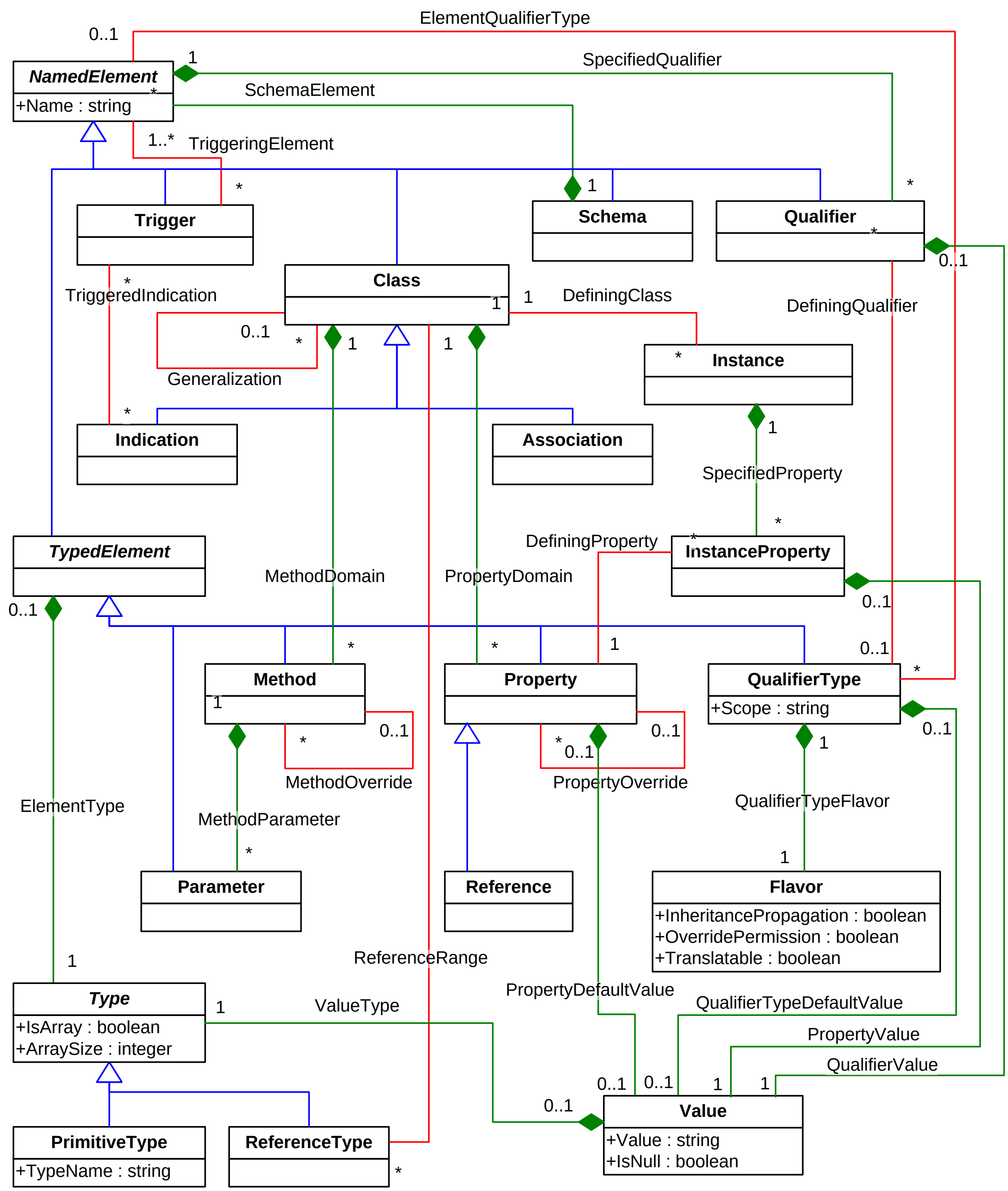



**Figure 2 – CIM Meta Schema**

NOTE: The CIM meta schema has been defined such that it can be defined as a CIM model provides a CIM model representing the CIM meta schema.

### 5.1.1 Formal Syntax used in Descriptions

In 5.1.2, the description of attributes and association ends of CIM meta-elements uses the following formal syntax defined in ABNF. Unless otherwise stated, the ABNF in this subclause has whitespace allowed. Further ABNF rules are defined in ANNEX A.

Descriptions of attributes use the `attribute-format` ABNF rule:

```
attribute-format = attr-name ":" attr-type ( "[" attr-multiplicity "]" )
    ; the format used to describe the attributes of CIM meta-elements

attr-name = IDENTIFIER
    ; the name of the attribute

attr-type = type
    ; the datatype of the attribute

type = "string"   ; a string of UCS characters of arbitrary length
     / "boolean"  ; a boolean value
     / "integer"  ; a signed 64-bit integer value

attr-multiplicity = cardinality-format
    ; the multiplicity of the attribute. The default multiplicity is 1
```

Descriptions of association ends use the `association-end-format` ABNF rule:

```
association-end-format = other-role ":" other-element "[" other-cardinality "]"
    ; the format used to describe association ends of associations
    ; between CIM meta-elements

other-role = IDENTIFIER
    ; the role of the association end (on this side of the relationship)
    ; that is referencing the associated meta-element

other-element = IDENTIFIER
    ; the name of the associated meta-element

other-cardinality = cardinality-format
    ; the cardinality of the associated meta-element

cardinality-format = positiveIntegerValue                    ; exactly that
                   / "*"                                     ; zero to any
                   / integerValue ".." positiveIntegerValue  ; min to max
                   / integerValue ".." "*"                   ; min to any
    ; format of a cardinality specification

integerValue = decimalDigit *decimalDigit                    ; no whitespace allowed

positiveIntegerValue = positiveDecimalDigit *decimalDigit    ; no whitespace allowed
```

### 5.1.2 CIM Meta-Elements

#### 5.1.2.1 NamedElement

Abstract class for CIM elements, providing the ability for an element to have a name.

Some kinds of elements provide the ability to have qualifiers specified on them, as described in subclasses of *NamedElement*.

Generalization: None

Non-default UML characteristics: isAbstract = True

Attributes:

- *Name* : string

  The name of the element. The format of the name is determined by subclasses of *NamedElement*.

  The names of elements shall be compared case-insensitively.

Association ends:

- *OwnedQualifier* : Qualifier [*] (composition *SpecifiedQualifier*, aggregating on its *OwningElement* end)

  The qualifiers specified on the element.

- *OwningSchema* : Schema [1] (composition *SchemaElement*, aggregating on its *OwningSchema* end)

  The schema owning the element.

- *Trigger* : Trigger [*] (association *TriggeringElement*)

  The triggers specified on the element.

- *QualifierType* : QualifierType [*] (association *ElementQualifierType*)

  The qualifier types implicitly defined on the element.

  Note: Qualifier types defined explicitly are not associated to elements; they are global in the CIM namespace.

---

**DEPRECATED**

The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

**DEPRECATED**

---

Additional constraints:

1) The value of *Name* shall not be Null.

#### 5.1.2.2 TypedElement

Abstract class for CIM elements that have a CIM data type.

Not all kinds of CIM data types may be used for all kinds of typed elements. The details are determined by subclasses of *TypedElement*.

Generalization: *NamedElement*

Non-default UML characteristics: *isAbstract* = True

Attributes: None

Association ends:

- *OwnedType* : Type [1] (composition *ElementType*, aggregating on its *OwningElement* end)

  The CIM data type of the element.

Additional constraints: None

#### 5.1.2.3 Type

Abstract class for any CIM data types, including arrays of such.

Generalizations: None

Non-default UML characteristics: *isAbstract* = True

Attributes:

- *IsArray* : boolean

  Indicates whether the type is an array type. For details on arrays, see 7.8.2.

- *ArraySize* : integer

  If the type is an array type, a non-Null value indicates the size of a fixed-size array, and a Null value indicates a variable-length array. For details on arrays, see 7.8.2.

Association ends:

- *OwningElement* : TypedElement [0..1] (composition *ElementType*, aggregating on its *OwningElement* end)
- *OwningValue* : Value [0..1] (composition *ValueType*, aggregating on its *OwningValue* end)

  The element that has a CIM data type.

Additional constraints:

1) The value of *IsArray* shall not be Null.
2) If the type is no array type, the value of *ArraySize* shall be Null.

   Equivalent OCL class constraint:

```
inv: self.IsArray = False
     implies self.ArraySize.IsNull()
```

3) A *Type* instance shall be owned by only one owner.

   Equivalent OCL class constraint:

```
inv: self.ElementType[OwnedType].OwningElement->size() +
     self.ValueType[OwnedType].OwningValue->size() = 1
```

#### 5.1.2.4 PrimitiveType

A CIM data type that is one of the intrinsic types defined in Table 2, excluding references.

Generalization: *Type*

Non-default UML characteristics: None

Attributes:

- *TypeName* : string

  The name of the CIM data type.

Association ends: None

Additional constraints:

1) The value of *TypeName* shall follow the formal syntax defined by the `dataType` ABNF rule in ANNEX A.
2) The value of *TypeName* shall not be Null.
3) This kind of type shall be used only for the following kinds of typed elements: *Method*, *Parameter*, ordinary *Property*, and *QualifierType*.

   Equivalent OCL class constraint:

```
inv: let e : _NamedElement =
       self.ElementType[OwnedType].OwningElement
     in
       e.oclIsTypeOf(Method) or
       e.oclIsTypeOf(Parameter) or
       e.oclIsTypeOf(Property) or
       e.oclIsTypeOf(QualifierType)
```

#### 5.1.2.5 ReferenceType

A CIM data type that is a reference, as defined in Table 2.

Generalization: *Type*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *ReferencedClass* : Class [1] (association *ReferenceRange*)

  The class referenced by the reference type.

Additional constraints:

1) This kind of type shall be used only for the following kinds of typed elements: *Parameter* and *Reference.*

   Equivalent OCL class constraint:

```
inv: let e : NamedElement = /* the typed element */
       self.ElementType[OwnedType].OwningElement
     in
       e.oclIsTypeOf(Parameter) or
       e.oclIsTypeOf(Reference)
```

2) When used for a *Reference*, the type shall not be an array.

Equivalent OCL class constraint:

```
inv: self.ElementType[OwnedType].OwningElement.
       oclIsTypeOf(Reference)
     implies
       self.IsArray = False
```

#### 5.1.2.6 Schema

Models a CIM schema. A CIM schema is a set of CIM classes with a single defining authority or owning organization.

Generalization: *NamedElement*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *OwnedElement* : NamedElement [*] (composition *SchemaElement*, aggregating on its *OwningSchema* end)

  The elements owned by the schema.

Additional constraints:

1) The value of the *Name* attribute shall follow the formal syntax defined by the `schemaName` ABNF rule in ANNEX A.
2) The elements owned by a schema shall be only of kind *Class*.

Equivalent OCL class constraint:

```
inv: self.SchemaElement[OwningSchema].OwnedElement.
       oclIsTypeOf(Class)
```

#### 5.1.2.7 Class

Models a CIM class. A CIM class is a common type for a set of CIM instances that support the same features (i.e., properties and methods). A CIM class models an aspect of a managed element.

Classes may be arranged in a generalization hierarchy that represents subtype relationships between classes. The generalization hierarchy is a rooted, directed graph and does not support multiple inheritance.

A class may have methods, which represent their behavior, and properties, which represent the data structure of its instances.

A class may participate in associations as the target of an association end owned by the association.

A class may have instances.

Generalization: *NamedElement*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *OwnedProperty* : Property [*] (composition *PropertyDomain*, aggregating on its *OwningClass* end)

  The properties owned by the class.
- *OwnedMethod* : Method [*] (composition *MethodDomain*, aggregating on its *OwningClass* end)

  The methods owned by the class.
- *ReferencingType* : ReferenceType [*] (association *ReferenceRange*)

  The reference types referencing the class.
- *SuperClass* : Class [0..1] (association *Generalization*)

  The superclass of the class.
- *SubClass* : Class [*] (association *Generalization*)

  The subclasses of the class.
- *Instance* : Instance [*] (association *DefiningClass*)

  The instances for which the class is their defining class.

Additional constraints:

1) The value of the *Name* attribute (i.e., the class name) shall follow the formal syntax defined by the `className` ABNF rule in ANNEX A.

   NOTE: The name of the schema containing the class is part of the class name.
2) The class name shall be unique within the schema owning the class.

#### 5.1.2.8 Property

Models a CIM property defined in a CIM class. A CIM property is the declaration of a structural feature of a CIM class, i.e., the data structure of its instances.

Properties are inherited to subclasses such that instances of the subclasses have the inherited properties in addition to the properties defined in the subclass. The combined set of properties defined in a class and properties inherited from superclasses is called the properties exposed by the class.

Classes that define a property without overriding an inherited property of the same name, expose two properties with that name. This is an undesirable situation since the resolution of property names to the actual properties is undefined in this document.

---

**DEPRECATED**

Within a single given schema (as defined in 5.1.2.6), the definition of properties without overriding inherited properties of the same name defined in a class of the same schema is deprecated. The deprecation only applies to the act of establishing that scenario, not necessarily to any schema elements that are involved.

**DEPRECATED**

---

Between an underlying schema (e.g., the DMTF published CIM schema) and a derived schema (e.g., a vendor schema), the definition of properties in the derived schema without overriding inherited properties of the same name defined in a class of the underlying schema may occur if both schemas are updated independently. Therefore, care should be exercised by the owner of the derived schema when moving to a new release of the underlying schema in order to avoid this situation.

A class defining a property may indicate that the property overrides an inherited property. In this case, the class exposes only the overriding property. The characteristics of the overriding property are formed by using the characteristics of the overridden property as a basis, changing them as defined in the overriding property, within certain limits as defined in section "Additional constraints".

If a property defines a default value, that default value shall be consistent with any initialization constraints for the property.

An initialization constraint limits the range of initial values of the property in new CIM instances. Initialization constraints for properties may be specified via the PropertyConstraint qualifier (see 5.6.3.39). Other specifications can additionally constrain the range of values for a property within a conformant implementation.

For example, management profiles may define initialization constraints, or operations may create new CIM instances with specific initial values.

The initial value of a property shall be conformant to all specified initialization constraints.

If no default value is defined for a property, and no value is provided at initialization, then the property will initially have no value, (i.e. it shall be Null.) Unless a property is specified to be Null at initialization time, an implementation may provide a value that is consistent with the property type and any initialization constraintsDefault values defined on properties in a class propagate to overriding properties in its subclasses. The value of the PropertyConstraint qualifier also propagates to overriding properties in subclasses, as defined in its qualifier type.

Generalization: *TypedElement*

Non-default UML characteristics: None

Attributes: None.

Association ends:

- *OwningClass* : Class [1] (composition *PropertyDomain*, aggregating on its *OwningClass* end)

  The class owning (i.e., defining) the property.

- *OverriddenProperty* : Property [0..1] (association *PropertyOverride*)

  The property overridden by this property.

- *OverridingProperty* : Property [*] (association *PropertyOverride*)

  The property overriding this property.

- *InstanceProperty* : *InstanceProperty* [*] (association *DefiningProperty*)

  A value of this property in an instance.

- *OwnedDefaultValue* : Value [0..1] (composition *PropertyDefaultValue*, aggregating on its *OwningProperty* end)

  The default value of the property declaration. A *Value* instance shall be associated if and only if a default value is defined on the property declaration.

Additional constraints:

1) The value of the *Name* attribute (i.e., the property name) shall follow the formal syntax defined by the `propertyName` ABNF rule in ANNEX A.

2) Property names shall be unique within its owning (i.e., defining) class.

3) An overriding property shall have the same name as the property it overrides.

Equivalent OCL class constraint:

```
inv: self.PropertyOverride[OverridingProperty]->
       size() = 1
     implies
     self.PropertyOverride[OverridingProperty].
       OverriddenProperty.Name.toUpper() =
     self.Name.toUpper()
```

NOTE: As a result of constraints 2) and 3), the set of properties exposed by a class may have duplicate names if a class defines a property with the same name as a property it inherits without overriding it.

4) The class owning an overridden property shall be a (direct or indirect) superclass of the class owning the overriding property.

5) For ordinary properties, the data type of the overriding property shall be the same as the data type of the overridden property.

Equivalent OCL class constraint:

```
inv: self.oclIsTypeOf(Meta_Property) and
       PropertyOverride[OverridingProperty]->
       size() = 1
     implies
       let pt :Type = /* type of property */
         self.ElementType[Element].Type
       in
       let opt : Type = /* type of overridden prop. */
         self.PropertyOverride[OverridingProperty].
         OverriddenProperty.Meta_ElementType[Element].Type
       in
       opt.TypeName.toUpper() = pt.TypeName.toUpper() and
       opt.IsArray   = pt.IsArray   and
       opt.ArraySize = pt.ArraySize
```

6) For references, the class referenced by the overriding reference shall be the same as, or a subclass of, the class referenced by the overridden reference.

7) A property shall have no more than one initialization constraint defined (either via its default value or via the *PropertyConstraint* qualifier, see 5.6.3.39).

8) A property shall have no more than one derivation constraint defined (via the *PropertyConstraint* qualifier, see 5.6.3.39).

#### 5.1.2.9 Method

Models a CIM method. A CIM method is the declaration of a behavioral feature of a CIM class, representing the ability for invoking an associated behavior.

The CIM data type of the method defines the declared return type of the method.

Methods are inherited to subclasses such that subclasses have the inherited methods in addition to the methods defined in the subclass. The combined set of methods defined in a class and methods inherited from superclasses is called the methods exposed by the class.

A class defining a method may indicate that the method overrides an inherited method. In this case, the class exposes only the overriding method. The characteristics of the overriding method are formed by using the characteristics of the overridden method as a basis, changing them as defined in the overriding method, within certain limits as defined in section "Additional constraints".

Classes that define a property without overriding an inherited property of the same name, expose two properties with that name. This is an undesirable situation since the resolution of property names to the actual properties is undefined in this document.

---

**DEPRECATED**

Within a single given schema (as defined in 5.1.2.6), the definition of properties without overriding inherited properties of the same name defined in a class of the same schema is deprecated. The deprecation only applies to the act of establishing that scenario, not necessarily to any schema elements that are involved.

**DEPRECATED**

---

Between an underlying schema (e.g., the DMTF published CIM schema) and a derived schema (e.g., a vendor schema), the definition of properties in the derived schema without overriding inherited properties of the same name defined in a class of the underlying schema may occur if both schemas are updated independently. Therefore, care should be exercised by the owner of the derived schema when moving to a new release of the underlying schema in order to avoid this situation.

Generalization: *TypedElement*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *OwningClass* : Class [1] (composition *MethodDomain*, aggregating on its *OwningClass* end)

  The class owning (i.e., defining) the method.

- *OwnedParameter* : Parameter [*] (composition *MethodParameter*, aggregating on its OwningMethod end)

  The parameters of the method. The return value of a method is not represented as a parameter.

- *OverriddenMethod* : *Method* [0..1] (association *MethodOverride*)

  The method overridden by this method.

- *OverridingMethod* : *Method* [*] (association *MethodOverride*)

  The method overriding this method.

Additional constraints:

1) The value of the *Name* attribute (i.e., the method name) shall follow the formal syntax defined by the `methodName` ABNF rule in ANNEX A.

2) Method names shall be unique within its owning (i.e., defining) class.

3) An overriding method shall have the same name as the method it overrides.

   Equivalent OCL class constraint:

```
inv: self.MethodOverride[OverridingMethod]->
       size() = 1
     implies
       self.MethodOverride[OverridingMethod].
         OverriddenMethod.Name.toUpper() =
       self.Name.toUpper()
```

NOTE: As a result of constraints 2) and 3), the set of methods exposed by a class may have duplicate names if a class defines a method with the same name as a method it inherits without overriding it.

4) The return type of a method shall not be an array.

Equivalent OCL class constraint:

```
inv: self.ElementType[Element].Type.IsArray = False
```

5) The class owning an overridden method shall be a superclass of the class owning the overriding method.

6) An overriding method shall have the same signature (i.e., parameters and return type) as the method it overrides.

Equivalent OCL class constraint:

```
inv: MethodOverride[OverridingMethod]->size() = 1
     implies
       let om : Method = /* overridden method */
         self.MethodOverride[OverridingMethod].
           OverriddenMethod
       in
       om.ElementType[Element].Type.TypeName.toUpper() =
         self.ElementType[Element].Type.TypeName.toUpper()
       and
       Set {1 .. om.MethodParameter[OwningMethod].
            OwnedParameter->size()}
       ->forAll( i /
         let omp : Parameter = /* parm in overridden method */
           om.MethodParameter[OwningMethod].OwnedParameter->
             asOrderedSet()->at(i)
         in
         let selfp : Parameter = /* parm in overriding method */
           self.MethodParameter[OwningMethod].OwnedParameter->
             asOrderedSet()->at(i)
         in
         omp.Name.toUpper() = selfp.Name.toUpper() and
         omp.ElementType[Element].Type.TypeName.toUpper() =
           selfp.ElementType[Element].Type.TypeName.toUpper()
       )
```

#### 5.1.2.10 Parameter

Models a CIM parameter. A CIM parameter is the declaration of a parameter of a CIM method. The return value of a method is not modeled as a parameter.

Generalization: *TypedElement*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *OwningMethod* : *Method* [1] (composition *MethodParameter*, aggregating on its *OwningMethod* end)

  The method owning (i.e., defining) the parameter.

Additional constraints:

1) The value of the *Name* attribute (i.e., the parameter name) shall follow the formal syntax defined by the `parameterName` ABNF rule in ANNEX A.

#### 5.1.2.11 Trigger

Models a CIM trigger. A CIM trigger is the specification of a rule on a CIM element that defines when the trigger is to be fired.

Triggers may be fired on the following occasions:

- On creation, deletion, modification, or access of CIM instances of ordinary classes and associations. The trigger is specified on the class in this case and applies to all instances.
- On modification, or access of a CIM property. The trigger is specified on the property in this case and applies to all instances.
- Before and after the invocation of a CIM method. The trigger is specified on the method in this case and applies to all invocations of the method.
- When a CIM indication is raised. The trigger is specified on the indication in this case and applies to all occurrences for when this indication is raised.

The rules for when a trigger is to be fired are specified with the *TriggerType* qualifier.

The firing of a trigger shall cause the indications to be raised that are associated to the trigger via *TriggeredIndication*.

Generalization: *NamedElement*

Non-default UML characteristics: None

Attributes: None

Association ends:

- Element : *NamedElement* [1..*] (association *TriggeringElement*)

  The CIM element on which the trigger is specified.

- Indication : *Indication* [*] (association *TriggeredIndication*)

  The CIM indications to be raised when the trigger fires.

Additional constraints:

1) The value of the *Name* attribute (i.e., the name of the trigger) shall be unique within the class, property, or method on which the trigger is specified.

2) Triggers shall be specified only on ordinary classes, associations, properties (including references), methods and indications.

   Equivalent OCL class constraint:

```
inv: let e : NamedElement = /* the element on which the trigger is specified*/
       self.TriggeringElement[Trigger].Element
     in
       e.oclIsTypeOf(Class) or
       e.oclIsTypeOf(Association) or
       e.oclIsTypeOf(Property) or
       e.oclIsTypeOf(Reference) or
       e.oclIsTypeOf(Method) or
       e.oclIsTypeOf(Indication)
```

#### 5.1.2.12 Indication

Models a CIM indication. An instance of a CIM indication represents an event that has occurred. If an instance of an indication is created, the indication is said to be *raised*. The event causing an indication to be raised may be that a trigger has fired, but other arbitrary events may cause an indication to be raised as well.

Generalization: *Class*

Non-default UML characteristics: None

Attributes: None

Association ends:

- *Trigger* : Trigger [*] (association *TriggeredIndication*)

  The triggers that when fired cause the indication to be raised.

Additional constraints:

1) An indication shall not own any methods.

   Equivalent OCL class constraint:

```
inv: self.MethodDomain[OwningClass].OwnedMethod->size() = 0
```

#### 5.1.2.13 Association

Models a CIM association. A CIM association is a special kind of CIM class that represents a relationship between two or more CIM classes. A CIM association owns its association ends (i.e., references). This allows for adding associations to a schema without affecting the associated classes.

Generalization: *Class*

Non-default UML characteristics: None

Attributes: None

Association ends: None

Additional constraints:

1) The superclass of an association shall be an association.

   Equivalent OCL class constraint:

```
inv: self.Generalization[SubClass].SuperClass->
       oclIsTypeOf(Association)
```

2) An association shall own two or more references.

   Equivalent OCL class constraint:

```
inv: self.PropertyDomain[OwningClass].OwnedProperty->
       select( p / p.oclIsTypeOf(Reference))->size() >= 2
```

3) The number of references exposed by an association (i.e., its arity) shall not change in its subclasses.

Equivalent OCL class constraint:

```
inv: self.PropertyDomain[OwningClass].OwnedProperty->
       select( p / p.oclIsTypeOf(Reference))->size() =
       self.Generalization[SubClass].SuperClass->
       PropertyDomain[OwningClass].OwnedProperty->
       select( p / p.oclIsTypeOf(Reference))->size()
```

#### 5.1.2.14 Reference

Models a CIM reference. A CIM reference is a special kind of CIM property that represents an association end, as well as a role the referenced class plays in the context of the association owning the reference.

Generalization: *Property*

Non-default UML characteristics: None

Attributes: None

Association ends: None

Additional constraints:

1) The value of the *Name* attribute (i.e., the reference name) shall follow the formal syntax defined by the `referenceName` ABNF rule in ANNEX A.

2) A reference shall be owned by an association (i.e., not by an ordinary class or by an indication).

   As a result of this, reference names do not need to be unique within any of the associated classes.

   Equivalent OCL class constraint:

```
inv: self.PropertyDomain[OwnedProperty].OwningClass.
       oclIsTypeOf(Association)
```

#### 5.1.2.15 Qualifier Type

Models the declaration of a CIM qualifier (i.e., a qualifier type). A CIM qualifier is meta data that provides additional information about the element on which the qualifier is specified.

The qualifier type is either explicitly defined in the CIM namespace, or implicitly defined on an element as a result of a qualifier that is specified on an element for which no explicit qualifier type is defined.

---

**DEPRECATED**

The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

**DEPRECATED**

---

Generalization: *TypedElement*

Non-default UML characteristics: None

Attributes:

- Scope : string [*]

  The scopes of the qualifier. The qualifier scopes determine to which kinds of elements a qualifier may be specified on. Each qualifier scope shall be one of the following keywords:

  – "any" - the qualifier may be specified on any qualifiable element.
  – "class" - the qualifier may be specified on any ordinary class.
  – "association" - the qualifier may be specified on any association.
  – "indication" - the qualifier may be specified on any indication.
  – "property" - the qualifier may be specified on any ordinary property.
  – "reference" - the qualifier may be specified on any reference.
  – "method" - the qualifier may be specified on any method.
  – "parameter" - the qualifier may be specified on any parameter.

  Qualifiers cannot be specified on qualifiers.

Association ends:

- *Flavor* : *Flavor* [1] (composition *QualifierTypeFlavor*, aggregating on its *QualifierType* end)

  The flavor of the qualifier type.

- *Qualifier* : *Qualifier* [*] (association *DefiningQualifier*)

  The specified qualifiers (i.e., usages) of the qualifier type.

- Element : *NamedElement* [0..1] (association *ElementQualifierType*)

  For implicitly defined qualifier types, the element on which the qualifier type is defined.

---

**DEPRECATED**

The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

**DEPRECATED**

---

Qualifier types defined explicitly are not associated to elements; they are global in the CIM namespace.

Additional constraints:

1) The value of the *Name* attribute (i.e., the name of the qualifier) shall follow the formal syntax defined by the `qualifierName` ABNF rule in ANNEX A.
2) The names of explicitly defined qualifier types shall be unique within the CIM namespace.

   NOTE: Unlike classes, qualifier types are not part of a schema, so name uniqueness cannot be defined at the definition level relative to a schema, and is instead only defined at the object level relative to a namespace.
3) The names of implicitly defined qualifier types shall be unique within the scope of the CIM element on which the qualifiers are specified.
4) Implicitly defined qualifier types shall agree in data type, scope, flavor and default value with any explicitly defined qualifier types of the same name.

**DEPRECATED**

The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

**DEPRECATED**

#### 5.1.2.16 Qualifier

Models the specification (i.e., usage) of a CIM qualifier on an element. A CIM qualifier is meta data that provides additional information about the element on which the qualifier is specified. The specification of a qualifier on an element defines a value for the qualifier on that element.

If no explicitly defined qualifier type exists with this name in the CIM namespace, the specification of a qualifier causes an implicitly defined qualifier type (i.e., a *QualifierType* element) to be created on the qualified element.

**DEPRECATED**

The concept of implicitly defined qualifier types is deprecated. Use explicitly defined qualifiers instead.

**DEPRECATED**

Generalization: *NamedElement*

Non-default UML characteristics: None

Attributes:

- *Value* : string [*]

  The value of the qualifier, in its string representation.

Association ends:

- *QualifierType* : QualifierType [1] (association *DefiningQualifier*)

  The qualifier type defining the characteristics of the qualifier.

- *OwningElement* : NamedElement [1] (composition *SpecifiedQualifier*, aggregating on its *OwningElement* end)

  The element on which the qualifier is specified.

Additional constraints:

1) The value of the *Name* attribute (i.e., the name of the qualifier) shall follow the formal syntax defined by the `qualifierName` ABNF rule in ANNEX A.

#### 5.1.2.17 Flavor

The specification of certain characteristics of the qualifier such as its value propagation from the ancestry of the qualified element, and translatability of the qualifier value.

Generalization: None

Non-default UML characteristics: None

Attributes:

- *InheritancePropagation* : boolean

  Indicates whether the qualifier value is to be propagated from the ancestry of an element in case the qualifier is not specified on the element.

- *OverridePermission* : boolean

  Indicates whether qualifier values propagated to an element may be overridden by the specification of that qualifier on the element.

- *Translatable* : boolean

  Indicates whether qualifier value is translatable.

Association ends:

- *QualifierType* : QualifierType [1]   (composition *QualifierTypeFlavor*, aggregating on its *QualifierType* end)

  The qualifier type defining the flavor.

Additional constraints:  None

#### 5.1.2.18 Instance

Models a CIM instance. A CIM instance is an instance of a CIM class that specifies values for a subset (including all) of the properties exposed by its defining class.

A CIM instance in a CIM server shall have exactly the properties exposed by its defining class.

A CIM instance cannot redefine the properties or methods exposed by its defining class and cannot have qualifiers specified.

Generalization:  None

Non-default UML characteristics:  None

Attributes:  None

Association ends:

- *OwnedPropertyValue* : PropertyValue [*]   (composition *SpecifiedProperty*, aggregating on its *OwningInstance* end)

  The property values specified by the instance.

- *DefiningClass* : Class [1]   (association *DefiningClass*)

  The defining class of the instance.

Additional constraints:

1) A particular property shall be specified at most once in a given instance.

#### 5.1.2.19 InstanceProperty

The definition of a property value within a CIM instance.

Generalization:  None

Non-default UML characteristics:  None

Attributes:

- *OwnedValue* :Value [1] (composition *PropertyValue*, aggregating on its *OwningInstanceProperty* end)

  The value of the property.

Association ends:

- *OwningInstance* : Instance [1] (composition *SpecifiedProperty*, aggregating on its *OwningInstance* end)

  The instance for which a property value is defined.

- *DefiningProperty* : PropertyValue [1] (association *DefiningProperty*)

  The declaration of the property for which a value is defined.

Additional constraints: None

#### 5.1.2.20 Value

A typed value, used in several contexts.

Generalization: None

Non-default UML characteristics: None

Attributes:

- *Value* : string [*]

  The scalar value or the array of values. Each value is represented as a string.

- *IsNull* : boolean

  The Null indicator of the value. If True, the value is Null. If False, the value is indicated through the Value attribute.

Association ends:

- *OwnedType* : Type [1] (composition *ValueType*, aggregating on its *OwningValue* end)

  The type of this value.

- *OwningProperty* : Property [0..1] (composition *PropertyDefaultValue*, aggregating on its *OwningProperty* end)

  A property declaration that defines this value as its default value.

- *OwningInstanceProperty* : InstanceProperty [0..1] (composition *PropertyValue*, aggregating on its *OwningInstanceProperty* end)

  A property defined in an instance that has this value.

- *OwningQualifierType* : QualifierType [0..1] (composition *QualifierTypeDefaultValue*, aggregating on its *OwningQualifierType* end)

  A qualifier type declaration that defines this value as its default value.

- *OwningQualifier* : Qualifier [0..1] (composition *QualifierValue*, aggregating on its *OwningQualifier* end)

  A qualifier defined on a schema element that has this value.

Additional constraints:

1) If the Null indicator is set, no values shall be specified.

Equivalent OCL class constraint:

```
inv: self.IsNull = True
     implies self.Value->size() = 0
```

2) If values are specified, the Null indicator shall not be set.

Equivalent OCL class constraint:

```
inv: self.Value->size() > 0
     implies self.IsNull = False
```

3) A Value instance shall be owned by only one owner.

Equivalent OCL class constraint:

```
inv: self.OwningProperty->size() +
     self.OwningInstanceProperty->size() +
     self.OwningQualifierType->size() +
     self.OwningQualifier->size() = 1
```

## 5.2 Data Types

Properties, references, parameters, and methods (that is, method return values) have a data type. These data types are limited to the intrinsic data types or arrays of such. Additional constraints apply to the data types of some elements, as defined in this document. Structured types are constructed by designing new classes. There are no subtype relationships among the intrinsic data types uint8, sint8, uint16, sint16, uint32, sint32, uint64, sint64, string, boolean, real32, real64, datetime, char16, and arrays of them. CIM elements of any intrinsic data type (including <classname> REF), and which are not further constrained in this document, may be initialized to NULL. NULL is a keyword that indicates the absence of value.

Table 2 lists the intrinsic data types and how they are interpreted.

**Table 2 – Intrinsic Data Types**

| Intrinsic Data Type | Interpretation |
|---|---|
| uint8 | Unsigned 8-bit integer |
| sint8 | Signed 8-bit integer |
| uint16 | Unsigned 16-bit integer |
| sint16 | Signed 16-bit integer |
| uint32 | Unsigned 32-bit integer |
| sint32 | Signed 32-bit integer |
| uint64 | Unsigned 64-bit integer |
| sint64 | Signed 64-bit integer |
| string | String of UCS characters as defined in 5.2.2 |
| boolean | Boolean |
| real32 | 4-byte floating-point value compatible with IEEE-754® Single format |
| real64 | 8-byte floating-point compatible with IEEE-754® Double format |
| datetime | A 7-bit ASCII string containing a date-time, as defined in 5.2.4 |
| <classname> ref | Strongly typed reference |
| char16 | UCS character in UCS-2 coded representation form, as defined in 5.2.3 |

### 5.2.1 UCS and Unicode

ISO/IEC 10646:2003 defines the *Universal Multiple-Octet Coded Character Set* (*UCS*). The Unicode Standard defines *Unicode*. This subclause 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 10646:2003 and its terminology. ISO/IEC 10646:2003 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+xxxxxx format defined in ISO/IEC 10646:2003. 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 *precomposed 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 10646:2003 references *The Unicode Standard, Version 5.2.0, 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 precomposed characters where possible, but not all characters of the UCS character repertoire can be represented as precomposed 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.

### 5.2.2 String Type

Non-Null string typed values shall contain zero or more UCS characters (see 5.2.1).

Implementations shall support a character repertoire for string typed values that is that defined by ISO/IEC 10646:2003 with its amendments ISO/IEC 10646:2003/Amd 1:2005 and ISO/IEC 10646:2003/Amd 2:2006 applied (this is the same character repertoire as defined by the Unicode Standard 5.0).

It is recommended that 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, Version 5.2.0, 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 10646:2003).

NOTE: Version 2.6.0 of this document introduced the requirement to support at least the character repertoire of ISO/IEC 10646:2003 with its amendments ISO/IEC 10646:2003/Amd 1:2005 and ISO/IEC 10646:2003/Amd 2:2006 applied. Previous versions of this document simply stated that the string type is a "UCS-2 string" without offering further details as to whether this was a definition of the character repertoire or a requirement on the usage of that coded representation form. UCS-2 does not support the character repertoire required in this subclause, and it does not satisfy the requirements of a number of countries, including the requirements of the Chinese national standard GB18030. UCS-2 was superseded by UTF-16 in Unicode 2.0 (released in 1996), although it is still in use today. For example, CIM clients that still use UCS-2 as an internal representation of string typed values will not be able to represent all characters that may be returned by a CIM server that supports the character repertoire required in this subclause.

### 5.2.3 Char16 Type

The char16 type is a 16-bit data entity. Non-Null char16 typed values shall contain one UCS character (see 5.2.1) in the coded representation form UCS-2 (defined in ISO/IEC 10646:2003).

**DEPRECATED**

Due to the limitations of UCS-2 (see 5.2.2), the char16 type is deprecated since version 2.6.0 of this document. Use the string type instead.

**DEPRECATED**

### 5.2.4 Datetime Type

The datetime type specifies a timestamp (point in time) or an interval. If it specifies a timestamp, the timezone offset can be preserved. In both cases, datetime specifies the date and time information with varying precision.

Datetime uses a fixed string-based format. The format for timestamps is:

```
yyyymmddhhmmss.mmmmmmsutc
```

The meaning of each field is as follows:

- `yyyy` is a 4-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 '+' (plus) or '–' (minus), indicating that the value is a timestamp, and indicating the sign of the UTC offset as described for the utc field.
- utc and s indicate the UTC offset of the time zone in which the time expressed by the other fields is the local time, including any effects of daylight savings time. The value of the utc field is the absolute of the offset of that time zone from UTC (Universal Coordinated Time) in minutes. The value of the s field is '+' (plus) for time zones east of Greenwich, and '–' (minus) for time zones west of Greenwich.

Timestamps are based on the proleptic Gregorian calendar, as defined in section 3.2.1, "The Gregorian calendar", of ISO 8601:2004.

Because datetime contains the time zone information, the original time zone can be reconstructed from the value. Therefore, the same timestamp can be specified using different UTC offsets by adjusting the hour and minutes fields accordingly.

Examples:

- Monday, January 25, 1998, at 1:30:15 PM EST (US Eastern Standard Time) is represented as 19980125133015.0000000-300. The same point in time is represented in the UTC time zone as 19980125183015.0000000+000.
- Monday, May 25, 1998, at 1:30:15 PM EDT (US Eastern Daylight Time) is represented as 19980525133015.0000000-240. The same point in time is represented in the German (summertime) time zone as 19980525193015.0000000+120.

An alternative representation of the same timestamp is `19980525183015.0000000+000`.

The format for intervals is as follows:

```
ddddddddhhmmss.mmmmmm:000
```

The meaning of each field is as follows:

- `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
```

For both timestamps and intervals, the field values shall be zero-padded so that the entire string is always 25 characters in length.

For both timestamps and intervals, 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`.

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 timezone 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 NULL in datetime comparison operations.

3) Overflow and underflow condition checking is performed on the result of the expression, as follows:

For timestamp results:

- A timestamp older than the oldest valid value in the timezone of the result produces an arithmetic underflow condition.
- A timestamp newer than the newest valid value in the timezone 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 instance, 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 timezone shall be used for any timestamp results.

NOTE: For most fields, asterisks can be used only with the granularity of the entire field.

Examples:

```
"20051003110000.000000+000" + "00000000002233.000000:000"
    evaluates to  "20051003112233.000000+000"

"20051003110000.******+000" + "00000000002233.000000:000"
    evaluates to  "20051003112233.******+000"

"20051003110000.******+000" + "00000000002233.00000*:000"
    evaluates to  "200510031122**.******+000"

"20051003110000.******+000" + "00000000002233.******:000"
    evaluates to   "200510031122**.******+000"

"20051003110000.******+000" + "00000000005959.******:000"
    evaluates to  "20051003******.******+000"

"20051003110000.******+000" + "000000000022**.******:000"
    evaluates to  "2005100311****.******+000"

"20051003112233.000000+000" - "00000000002233.000000:000"
    evaluates to  "20051003110000.000000+000"

"20051003112233.******+000" - "00000000002233.000000:000"
    evaluates to  "20051003110000.******+000"

"20051003112233.******+000" - "00000000002233.00000*:000"
    evaluates to  "20051003110000.******+000"

"20051003112233.******+000" - "00000000002232.******:000"
    evaluates to  "200510031100**.******+000"
```

```
"20051003112233.******+000" - "00000000002233.******:000"
    evaluates to  "20051003******.******+000"

"20051003060000.000000-300" + "00000000002233.000000:000"
    evaluates to  "20051003112233.000000+000"

"20051003060000.******-300" + "00000000002233.000000:000"
    evaluates to  "20051003112233.******+000"

"000000000011**.******:000" * 60
    evaluates to  "0000000011****.******:000"

60 times adding up "000000000011**.******:000"
    evaluates to  "0000000011****.******:000"

"20051003112233.000000+000" = "20051003112233.000000+000"
    evaluates to True

"20051003122233.000000+060" = "20051003112233.000000+000"
    evaluates to True

"20051003112233.******+000" = "20051003112233.******+000"
    evaluates to Null (uncertain)

"20051003112233.******+000" = "200510031122**.******+000"
    evaluates to Null (uncertain)

"20051003112233.******+000" = "20051003112234.******+000"
    evaluates to False

"20051003112233.******+000" < "20051003112234.******+000"
    evaluates to True

"20051003112233.5*****+000" < "20051003112233.******+000"
    evaluates to Null (uncertain)
```

A datetime 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 within the CIM infrastructure.

Within these valid ranges, some values are defined as reserved. Values from these reserved ranges shall not be interpreted as points in time or durations.

Within these reserved ranges, some values have special meaning. The CIM schema should not define additional class-specific special values from the reserved range.

The valid and reserved ranges and the special values are defined as follows:

- For timestamp values:

| | |
|---|---|
| Oldest valid timestamp: | `"00000101000000.000000+720"` |
| | Reserved range (1 million values) |
| Oldest useable timestamp: | `"00000101000001.000000+720"` |
| | Range interpreted as points in time |
| Youngest useable timestamp: | `"99991231115959.999998-720"` |
| | Reserved range (1 value) |
| Youngest valid timestamp: | `"99991231115959.999999-720"` |

Special values in the reserved ranges:

| | |
|---|---|
| "Now": | `"00000101000000.000000+720"` |
| "Infinite past": | `"00000101000000.999999+720"` |
| "Infinite future": | `"99991231115959.999999-720"` |

- For interval values:

| | |
|---|---|
| Smallest valid and useable interval: | `"00000000000000.000000:000"` |
| | Range interpreted as durations |
| Largest useable interval: | `"99999999235958.999999:000"` |
| | Reserved range (1 million values) |
| Largest valid interval: | `"99999999235959.999999:000"` |

Special values in reserved range:

| | |
|---|---|
| "Infinite duration": | `"99999999235959.000000:000"` |

### 5.2.5 Indicating Additional Type Semantics with Qualifiers

Because counter and gauge types are actually simple integers with specific semantics, they are not treated as separate intrinsic types. Instead, qualifiers must be used to indicate such semantics when properties are declared. The following example merely suggests how this can be done; the qualifier names chosen are not part of this standard:

```
class ACME_Example
{
      [Counter]
   uint32 NumberOfCycles;

      [Gauge]
   uint32 MaxTemperature;

      [OctetString, ArrayType("Indexed")]
   uint8 IPAddress[10];
};
```

For documentation purposes, implementers are permitted to introduce such arbitrary qualifiers. The semantics are not enforced.

### 5.2.6 Comparison of Values

This subclause 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.

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 754-1985.

Values of the string and char16 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. As a result, comparisons between a char16 typed value and a string typed value are valid.

In order to minimize the processing involved in UCS normalization, string and char16 typed values should be stored and transmitted in Normalization Form C (NFC, see 5.2.2) where possible, which allows skipping the costly normalization when comparing the strings.

This document does not define an order between values of the string and char16 datatypes, since UCS ordering rules may be compute intensive and their usage should be decided on a case by case basis. The ordering of the "Common Template Table" defined in ISO/IEC 14651:2007 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 may be sufficient, for example when no human consumption of the ordering result is needed.

Values of schema elements qualified as octetstrings shall be compared for equality and ordering based on the sequence of octets they represent. As a result, comparisons across different octetstring representations (as defined in 5.6.3.35) are valid. Two sequences of octets 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, 2004 Edition.

Two values of the reference datatype shall be considered equal if they resolve to the same CIM object in the same namespace. 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 duration 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 timezone offsets are considered equal.

Two values of compatible datatypes that both are Null shall be considered equal. This document does not define an order between two values of compatible datatypes where one is Null, and the other is not Null.

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.

## 5.3 Backwards Compatibility

This subclause defines the general rules for backwards compatibility between CIM client, CIM server and CIM listener across versions.

The consequencs of these rules for CIM schema definitions are defined in 5.4. The consequences of these rules for other areas covered by DMTF (such as protocols or management profiles) are defined in the DMTF documents covering such other areas. The consequences of these rules for areas covered by business entities other than DMTF (such as APIs or tools) should be defined by these business entities.

Backwards compatibility between CIM client, CIM server and CIM listener is defined from a CIM client application perspective in relation to a CIM implementation:

- Newer compatible CIM implementations need to work with unchanged CIM client applications.

For the purposes of this rule, a "CIM client application" assumes the roles of CIM client and CIM listener, and a "CIM implementation" assumes the role of a CIM server. As a result, newer compatible CIM servers need to work with unchanged CIM clients and unchanged CIM listeners.

For the purposes of this rule, "newer compatible CIM implementations" have implemented DMTF specifications that have increased only the minor or update version indicators, but not the major version indicator, and that are relevant for the interface between CIM implementation and CIM client application.

Newer compatible CIM implementations may also have implemented newer compatible specifications of business entities other than DMTF that are relevant for the interface between CIM implementation and CIM client application (for example, vendor extension schemas); how that translates to version indicators of these specifications is left to the owning business entity.

## 5.4 Supported Schema Modifications

This subclause lists typical modifications of schema definitions and qualifier type declarations and defines their compatibility. Such modifications might be introduced into an existing CIM environment by upgrading the schema to a newer schema version. However, any rules for the modification of schema related objects (i.e., classes and qualifier types) in a CIM server are outside of the scope of this document. Specifications dealing with modification of schema related objects in a CIM server should define such rules and should consider the compatibility defined in this subclause.

Table 3 lists modifications of an existing schema definition (including an empty schema). The compatibility of the modification is indicated for CIM clients that utilize the modified element, and for a CIM server that implements the modified element. Compatibility for a CIM server that utilizes the modified element (e.g., via so called "up-calls") is the same as for a CIM client that utilizes the modified element.

The compatibility for CIM clients as expressed in Table 3 assumes that the CIM client remains unchanged and is exposed to a CIM server that was updated to fully reflect the schema modification.

The compatibility for CIM servers as expressed in Table 3 assumes that the CIM server remains unchanged but is exposed to the modified schema that is loaded into the CIM namespace being serviced by the CIM server.

Compatibility is stated as follows:

- Transparent – the respective component does not need to be changed in order to properly deal with the modification
- Not transparent – the respective component needs to be changed in order to properly deal with the modification

Schema modifications qualified as transparent for both CIM clients and CIM servers are allowed in a minor version update of the schema. Any other schema modifications are allowed only in a major version update of the schema.

The schema modifications listed in Table 3 cover simple cases, which may be combined to yield more complex cases. For example, a typical schema change is to move existing properties or methods into a new superclass. The compatibility of this complex schema modification can be determined by concatenating simple schema modifications listed in Table 3, as follows:

1) SM1: Adding a class to the schema:

   The new superclass gets added as an empty class with (yet) no superclass

2) SM3: Inserting an existing class that defines no properties or methods into an inheritance hierarchy of existing classes:

   The new superclass gets inserted into an inheritance hierarchy

3) SM8: Moving an existing property from a class to one of its superclasses (zero or more times)

   Properties get moved to the newly inserted superclass

4) SM12: Moving a method from a class to one of its superclasses (zero or more times)

   Methods get moved to the newly inserted superclass

The resulting compatibility of this complex schema modification for CIM clients is transparent, since all these schema modifications are transparent. Similarly, the resulting compatibility for CIM servers is transparent for the same reason.

Some schema modifications cause other changes in the schema to happen. For example, the removal of a class causes any associations or method parameters that reference that class to be updated in some way.

2076 **Table 3 – Compatibility of Schema Modifications**

| Schema Modification | Compatibility for CIM clients | Compatibility for CIM servers | Allowed in a Minor Version Update of the Schema |
|---|---|---|---|
| SM1: Adding a class to the schema. The new class may define an existing class as its superclass | Transparent.<br>It is assumed that any CIM clients that examine classes are prepared to deal with new classes in the schema and with new subclasses of existing classes | Transparent | Yes |
| SM2: Removing a class from the schema | Not transparent | Not transparent | No |
| SM3: Inserting an existing class that defines no properties or methods into an inheritance hierarchy of existing classes | Transparent.<br>It is assumed that any CIM clients that examine classes are prepared to deal with such inserted classes | Transparent | Yes |
| SM4: Removing an abstract class that defines no properties or methods from an inheritance hierarchy of classes, without removing the class from the schema | Not transparent | Transparent | No |
| SM5: Removing a concrete class that defines no properties or methods from an inheritance hierarchy of classes, without removing the class from the schema | Not transparent | Not transparent | No |
| SM6: Adding a property to an existing class that is not overriding a property. The property may have a non-Null default value | Transparent<br>It is assumed that CIM clients are prepared to deal with any new properties in classes and instances. | Transparent<br>If the CIM server uses the factory approach (1) to populate the properties of any instances to be returned, the property will be included in any instances of the class with its default value. Otherwise, the (unchanged) CIM server will not include the new property in any instances of the class, and a CIM client that knows about the new property will interpret it as having the Null value. | Yes |

| Schema Modification | Compatibility for CIM clients | Compatibility for CIM servers | Allowed in a Minor Version Update of the Schema |
|---|---|---|---|
| SM7: Adding a property to an existing class that is overriding a property. The overriding property does not define a type or qualifiers such that the overridden property is changed in a non-transparent way, as defined in schema modifications 17, xx. The overriding property may define a default value other than the overridden property | Transparent | Transparent | Yes |
| SM8: Moving an existing property from a class to one of its superclasses | Transparent.<br>It is assumed that any CIM clients that examine classes are prepared to deal with such moved properties. For CIM clients that deal with instances of the class from which the property is moved away, this change is transparent, since the set of properties in these instances does not change. For CIM clients that deal with instances of the superclass to which the property was moved, this change is also transparent, since it is an addition of a property to that superclass (see SM6). | Transparent.<br>For the implementation of the class from which the property is moved away, this change is transparent. For the implementation of the superclass to which the property is moved, this change is also transparent, since it is an addition of a property to that superclass (see SM6). | Yes |
| SM9: Removing a property from an existing class, without adding it to one of its superclasses | Not transparent | Not transparent | No |
| SM10: Adding a method to an existing class that is not overriding a method | Transparent<br>It is assumed that any CIM clients that examine classes are prepared to deal with such added methods. | Transparent<br>It is assumed that a CIM server is prepared to return an error to CIM clients indicating that the added method is not implemented. | Yes |

| Schema Modification | Compatibility for CIM clients | Compatibility for CIM servers | Allowed in a Minor Version Update of the Schema |
|---|---|---|---|
| SM11: Adding a method to an existing class that is overriding a method. The overriding method does not define a type or qualifiers on the method or its parameters such that the overridden method or its parameters are changed in an non-transparent way, as defined in schema modifications 16, xx | Transparent | Transparent | Yes |
| SM12: Moving a method from a class to one of its superclasses | Transparent<br>It is assumed that any CIM clients that examine classes are prepared to deal with such moved methods. For CIM clients that invoke methods on the class or instances thereof from which the method is moved away, this change is transparent, since the set of methods that are invocable on these classes or their instances does not change. For CIM clients that invoke methods on the superclass or instances thereof to which the property was moved, this change is also transparent, since it is an addition of a method to that superclass (see SM10) | Transparent<br>For the implementation of the class from which the method is moved away, this change is transparent. For the implementation of the class from which the method is moved away, this change is transparent. For the implementation of the superclass to which the method is moved, this change is also transparent, since it is an addition of a method to that superclass (see SM10). | Yes |
| SM13: Removing a method from an existing class, without adding it to one of its superclasses | Not transparent | Not transparent | No |
| SM14: Adding a parameter to an existing method | Not transparent | Not transparent | No |
| SM15: Removing a parameter from an existing method | Not transparent | Not transparent | No |
| SM16: Changing the non-reference type of an existing method parameter, method (i.e., its return value), or ordinary property | Not transparent | Not transparent | No |

| Schema Modification | Compatibility for CIM clients | Compatibility for CIM servers | Allowed in a Minor Version Update of the Schema |
|---|---|---|---|
| SM17: Changing the class referenced by a reference in an association to a subclass of the previously referenced class | Transparent | Not Transparent | No |
| SM18: Changing the class referenced by a reference in an association to a superclass of the previously referenced class | Not Transparent | Not Transparent | No |
| SM19: Changing the class referenced by a reference in an association to any class other than a subclass or superclass of the previously referenced class | Not Transparent | Not Transparent | No |
| SM20: Changing the class referenced by a method input parameter of reference type to a subclass of the previously referenced class | Not Transparent | Transparent | No |
| SM21: Changing the class referenced by a method input parameter of reference type to a superclass of the previously referenced class | Transparent | Not Transparent | No |
| SM22: Changing the class referenced by a method input parameter of reference type to any class other than a subclass or superclass of the previously referenced class | Not Transparent | Not Transparent | No |
| SM23: Changing the class referenced by a method output parameter or method return value of reference type to a subclass of the previously referenced class | Transparent | Not Transparent | No |

| Schema Modification | Compatibility for CIM clients | Compatibility for CIM servers | Allowed in a Minor Version Update of the Schema |
|---|---|---|---|
| SM24: Changing the class referenced by a method output parameter or method return value of reference type to a superclass of the previously referenced class | Not Transparent | Transparent | No |
| SM25: Changing the class referenced by a method output parameter or method return value of reference type to any class other than a subclass or superclass of the previously referenced class | Not Transparent | Not Transparent | No |
| SM26: Changing a class between ordinary class, association or indication | Not transparent | Not transparent | No |
| SM27: Reducing or increasing the arity of an association (i.e., increasing or decreasing the number of references exposed by the association) | Not transparent | Not transparent | No |
| SM28: Changing the effective value of a qualifier on an existing schema element | As defined in the qualifier description in 5.6 | As defined in the qualifier description in 5.6 | Yes, if transparent for both CIM clients and CIM servers, otherwise No |

1) Factory approach to populate the properties of any instances to be returned:

   Some CIM server architectures (e.g., CMPI-based CIM providers) support factory methods that create an internal representation of a CIM instance by inspecting the class object and creating property values for all properties exposed by the class and setting those values to their class defined default values. This delegates the knowledge about newly added properties to the schema definition of the class and will return instances that are compliant to the modified schema without changing the code of the CIM server. A subsequent release of the CIM server can then start supporting the new property with more reasonable values than the class defined default value.

Table 4 lists modifications of qualifier types. The compatibility of the modification is indicated for an existing schema. Compatibility for CIM clients or CIM servers is determined by Table 4 (in any modifications that are related to qualifier values).

The compatibility for a schema as expressed in Table 4 assumes that the schema remains unchanged but is confronted with a qualifier type declaration that reflects the modification.

Compatibility is stated as follows:

- Transparent – the schema does not need to be changed in order to properly deal with the modification
- Not transparent – the schema needs to be changed in order to properly deal with the modification

CIM supports extension schemas, so the actual usage of qualifiers in such schemas is by definition unknown and any possible usage needs to be assumed for compatibility considerations.

**Table 4 – Compatibility of Qualifier Type Modifications**

| Qualifier Type Modification | Compatibility for Existing Schema | Allowed in a Minor Version Update of the Schema |
|---|---|---|
| QM1: Adding a qualifier type declaration | Transparent | Yes |
| QM2: Removing a qualifier type declaration | Not transparent | No |
| QM3: Changing the data type or array-ness of an existing qualifier type declaration | Not transparent | No |
| QM4: Adding an element type to the scope of an existing qualifier type declaration, without adding qualifier value specifications to the element type added to the scope | Transparent | Yes |
| QM5: Removing an element type from the scope of an existing qualifier type declaration | Not transparent | No |
| QM6: Changing the inheritance flavors of an existing qualifier type declaration from ToSubclass DisableOverride to ToSubclass EnableOverride | Transparent | Yes |
| QM7: Changing the inheritance flavors of an existing qualifier type declaration from ToSubclass EnableOverride to ToSubclass DisableOverride | Not transparent | No |
| QM8: Changing the inheritance flavors of an existing qualifier type declaration from Restricted to ToSubclass EnableOverride | Transparent (generally) | Yes, if examination of the specific change reveals its compatibility |
| QM9: Changing the inheritance flavors of an existing qualifier type declaration from ToSubclass EnableOverride to Restricted | Transparent (generally) | Yes, if examination of the specific change reveals its compatibility |
| QM10: Changing the inheritance flavors of an existing qualifier type declaration from Restricted to ToSubclass DisableOverride | Not transparent (generally) | No, unless examination of the specific change reveals its compatibility |
| QM11: Changing the inheritance flavors of an existing qualifier type declaration from ToSubclass DisableOverride to Restricted | Transparent (generally) | Yes, if examination of the specific change reveals its compatibility |
| QM12: Changing the Translatable flavor of an existing qualifier type declaration | Transparent | Yes |

### 5.4.1 Schema Versions

Schema versioning is described in DSP4004. Versioning takes the form m.n.u, where:

- m = major version identifier in numeric form
- n = minor version identifier in numeric form
- u = update (errata or coordination changes) in numeric form

The usage rules for the Version qualifier in 5.6.3.55 provide additional information.

Classes are versioned in the CIM schemas. The Version qualifier for a class indicates the schema release of the last change to the class. Class versions in turn dictate the schema version. A major version change for a class requires the major version number of the schema release to be incremented. All class versions must be at the same level or a higher level than the schema release because classes and models that differ in minor version numbers shall be backwards-compatible. In other words, valid instances shall continue to be valid if the minor version number is incremented. Classes and models that differ in major version numbers are not backwards-compatible. Therefore, the major version number of the schema release shall be incremented.

Table 5 lists modifications to the CIM schemas in final status that cause a major version number change. Preliminary models are allowed to evolve based on implementation experience. These modifications change application behavior and/or customer code. Therefore, they force a major version update and are discouraged. Table 5 is an exhaustive list of the possible modifications based on current CIM experience and knowledge. Items could be added as new issues are raised and CIM standards evolve.

Alterations beyond those listed in Table 5 are considered interface-preserving and require the minor version number to be incremented. Updates/errata are not classified as major or minor in their impact, but they are required to correct errors or to coordinate across standards bodies.

2121 **Table 5 – Changes that Increment the CIM Schema Major Version Number**

| Description | Explanation or Exceptions |
|---|---|
| Class deletion | |
| Property deletion or data type change | |
| Method deletion or signature change | |
| Reorganization of values in an enumeration | The semantics and mappings of an enumeration cannot change, but values can be added in unused ranges as a minor change or update. |
| Movement of a class upwards in the inheritance hierarchy; that is, the removal of superclasses from the inheritance hierarchy | The removal of superclasses deletes properties or methods. New classes can be inserted as superclasses as a minor change or update. Inserted classes shall not change keys or add required properties. |
| Addition of Abstract, Indication, or Association qualifiers to an existing class | |
| Change of an association reference downward in the object hierarchy to a subclass or to a different part of the hierarchy | The change of an association reference to a subclass can invalidate existing instances. |
| Addition or removal of a Key or Weak qualifier | |
| Addition of the Required qualifier to a method input parameter or a property that may be written | Changing to require a non-Null value to be passed to an input parameter or to be written to a property may break existing CIM clients that pass Null under the prior definition.<br>An addition of the Required qualifier to method output parameters, method return values and properties that may only be read is considered a compatible change, as CIM clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the CIM server, as defined in 5.6.3.43.<br>The description of an existing schema element that added the Required qualifier in a revision of the schema should indicate the schema version in which this change was made, as defined in 5.6.3.43. |
| Removal of the Required qualifier from a method output parameter, a method (i.e., its return value) or a property that may be read | Changing to no longer guarantee a non-Null value to be returned by an output parameter, a method return value, or a property that may be read may break existing CIM clients that relied on the prior guarantee.<br>A removal of the Required qualifier from method input parameters and properties that may only be written is a compatible change, as CIM clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the CIM server, as defined in 5.6.3.43.<br>The description of an existing schema element that removed the Required qualifier in a revision of the schema should indicate the schema version in which this change was made, as defined in 5.6.3.43. |
| Decrease in MaxLen, decrease in MaxValue, increase in MinLen, or increase in MinValue | Decreasing a maximum or increasing a minimum invalidates current data. The opposite change (increasing a maximum) results in truncated data, where necessary. |
| Decrease in Max or increase in Min cardinalities | |
| Addition or removal of Override qualifier | There is one exception. An Override qualifier can be added if a property is promoted to a superclass, and it is necessary to maintain the specific qualifiers and descriptions in the original subclass. In this case, there is no change to existing instances. |

| Description | Explanation or Exceptions |
|---|---|
| Change in the following qualifiers: In/Out, Units | |

## 5.5 Class Names

Fully-qualified class names are in the form <schema name>_<class name>. An underscore is used as a delimiter between the <schema name> and the <class name>. The delimiter cannot appear in the <schema name> although it is permitted in the <class name>.

The format of the fully-qualified name allows the scope of class names to be limited to a schema. That is, the schema name is assumed to be unique, and the class name is required to be unique only within the schema. The isolation of the schema name using the underscore character allows user interfaces conveniently to strip off the schema when the schema is implied by the context.

The following are examples of fully-qualified class names:

- CIM_ManagedSystemElement: the root of the CIM managed system element hierarchy
- CIM_ComputerSystem: the object representing computer systems in the CIM schema
- CIM_SystemComponent: the association relating systems to their components
- Win32_ComputerSystem: the object representing computer systems in the Win32 schema

## 5.6 Qualifiers

Qualifiers are named and typed values that provide information about CIM elements. Since the qualifier values are on CIM elements and not on CIM instances, they are considered to be meta-data.

Subclause 5.6.1 describes the concept of qualifiers, independently of their representation in MOF. For their representation in MOF, see 7.7.

Subclauses 5.6.2, 5.6.3, and 5.6.4 describe the meta, standard, and optional qualifiers, respectively. Any qualifier type declarations with the names of these qualifiers shall have the name, type, scope, flavor, and default value defined in these subclauses.

Subclause 5.6.5 describes user-defined qualifiers.

Subclause 5.6.6 describes how the MappingString qualifier can be used to define mappings between CIM and other information models.

### 5.6.1 Qualifier Concept

#### 5.6.1.1 Qualifier Value

Any qualifiable CIM element (i.e., classes including associations and indications, properties including references, methods and parameters) shall have a particular set of qualifier values, as follows. A qualifier shall have a value on a CIM element if that kind of CIM element is in the scope of the qualifier, as defined in 5.6.1.3. If a kind of CIM element is in the scope of a qualifier, the qualifier is said to be an applicable qualifier for that kind of CIM element and for a specific CIM element of that kind.

Any applicable qualifier may be specified on a CIM element. When an applicable qualifier is specified on a CIM element, the qualifier shall have an explicit value on that CIM element. When an applicable qualifier is not specified on a CIM element, the qualifier shall have an assumed value on that CIM element, as defined in 5.6.1.5.

The value specified for a qualifier shall be consistent with the data type defined by its qualifier type.

There shall not be more than one qualifier with the same name specified on any CIM element.

#### 5.6.1.2 Qualifier Type

A qualifier type defines name, data type, scope, flavor and default value of a qualifier, as follows:

The name of a qualifier is a string that shall follow the formal syntax defined by the `qualifierName` ABNF rule in ANNEX A.

The data type of a qualifier shall be one of the intrinsic data types defined in Table 2, including arrays of such, excluding references and arrays thereof. If the data type is an array type, the array shall be an indexed variable length array, as defined in 7.8.2.

The scope of a qualifier determines which kinds of CIM elements have a value of that qualifier, as defined in 5.6.1.3.

The flavor of a qualifier determines propagation to subclasses, override permissions, and translatability, as defined in 5.6.1.4.

The default value of a qualifier is used to determine the effective value of qualifiers that are not specified on a CIM element, as defined in 5.6.1.5.

There shall not exist more than one qualifier type object with the same name in a CIM namespace. Qualifier types are not part of a schema; therefore name uniqueness of qualifiers cannot be defined within the boundaries of a schema (like it is done for class names).

#### 5.6.1.3 Qualifier Scope

The scope of a qualifier determines which kinds of CIM elements have a value for that qualifier.

The scope of a qualifier shall be one or more of the scopes defined in Table 6, except for scope (Any) whose specification shall not be combined with the specification of the other scopes. Qualifiers cannot be specified on qualifiers.

**Table 6 – Defined Qualifier Scopes**

| Qualifier Scope | Qualifier may be specified on … |
|---|---|
| Class | ordinary classes |
| Association | Associations |
| Indication | Indications |
| Property | ordinary properties |
| Reference | References |
| Method | Methods |
| Parameter | method parameters |
| Any | any of the above |

#### 5.6.1.4 Qualifier Flavor

The flavor of a qualifier determines propagation of its value to subclasses, override permissions of the propagated value, and translatability of the value.

The flavor of a qualifier shall be zero or more of the flavors defined in Table 7, subject to further restrictions defined in this subclause.

**Table 7 – Defined Qualifier Flavors**

| Qualifier Flavor | If the flavor is specified, ... |
|---|---|
| ToSubclass | propagation to subclasses is enabled (the implied default) |
| Restricted | propagation to subclasses is disabled |
| EnableOverride | if propagation to subclasses is enabled, override permission is granted (the implied default) |
| DisableOverride | if propagation to subclasses is enabled, override permission is not granted |
| Translatable | specification of localized qualifiers is enabled (by default it is disabled) |

Flavor (ToSubclass) and flavor (Restricted) shall not be specified both on the same qualifier type. If none of these two flavors is specified on a qualifier type, flavor (ToSubclass) shall be the implied default.

If flavor (Restricted) is specified, override permission is meaningless. Thus, flavor (EnableOverride) and flavor (DisableOverride) should not be specified and are meaningless if specified.

Flavor (EnableOverride) and flavor (DisableOverride) shall not be specified both on the same qualifier type. If none of these two flavors is specified on a qualifier type, flavor (EnableOverride) shall be the implied default.

This results in three meaningful combinations of these flavors:

- Restricted – propagation to subclasses is disabled
- EnableOverride – propagation to subclasses is enabled and override permission is granted
- DisableOverride – propagation to subclasses is enabled and override permission is not granted

If override permission is not granted for a qualifier type, then for a particular CIM element in the scope of that qualifier type, a qualifier with that name may be specified multiple times in the ancestry of its class, but each occurrence shall specify the same value. This semantics allows the qualifier to change its effective value at most once along the ancestry of an element.

If flavor (Translatable) is specified on a qualifier type, the specification of localized qualifiers shall be enabled for that qualifier, otherwise it shall be disabled. Flavor (Translatable) shall be specified only on qualifier types that have data type string or array of strings. For details, see 5.6.1.6.

#### 5.6.1.5 Effective Qualifier Values

When there is a qualifier type defined for a qualifier, and the qualifier is applicable but not specified on a CIM element, the CIM element shall have an assumed value for that qualifier. This assumed value is called the effective value of the qualifier.

The effective value of a particular qualifier on a given CIM element shall be determined as follows:

If the qualifier is specified on the element, the effective value is the value of the specified qualifier. In MOF, qualifiers may be specified without specifying a value, in which case a value is implied, as described in 7.7.

If the qualifier is not specified on the element and propagation to subclasses is disabled, the effective value is the default value defined on the qualifier type declaration.

If the qualifier is not specified on the element and propagation to subclasses is enabled, the effective value is the value of the nearest like-named qualifier that is specified in the ancestry of the element. If the qualifier is not specified anywhere in the ancestry of the element, the effective value is the default value defined on the qualifier type declaration.

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, its superclass is its ancestor element. If the class does not have a superclass, it has no ancestor.
- For an overiding 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 a 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.

#### 5.6.1.6 Localized Qualifiers

Localized qualifiers allow the specification of qualifier values in a specific language.

---

**DEPRECATED**

Localized qualifiers and the flavor (Translatable) as described in this subclause have been deprecated. The usage of localized qualifiers is discouraged.

**DEPRECATED**

---

The qualifier type on which flavor (Translatable) is specified, is called the base qualifier of its localized qualifiers.

The name of any localized qualifiers shall conform to the following formal syntax defined in ABNF:

```
localized-qualifier-name = qualifier-name "_" locale

locale = language-code "_" country code
         ; the locale of the localized qualifier
```

Where:

`qualifier-name` is the name of the base qualifier of the localized qualifier

`language-code` is a language code as defined in ISO 639-1:2002, ISO 639-2:1996, or ISO 639-3:2007

`country-code` is a country code as defined in ISO 3166-1:2006, ISO 3166-2:2007, or ISO 3166-3:1999

EXAMPLE:

For the base qualifier named Description, the localized qualifier for Mexican Spanish language is named Description_es_MX.

The string value of a localized qualifier shall be a translation of the string value of its base qualifier from the language identified by the locale of the base qualifier into the language identified by the locale specified in the name of the localized qualifier.

For MOF, the locale of the base qualifier shall be the locale defined by the preceding #pragma locale directive.

For any localized qualifiers specified on a CIM element, a qualifier type with the same name (i.e., including the locale suffix) may be declared. If such a qualifier type is declared, its type, scope, flavor and default value shall match the type, scope, flavor and default value of the base qualifier. If such a qualifier type is not declared, it is implied from the qualifier type declaration of the base qualifier, with unchanged type, scope, flavor and default value.

### 5.6.2 Meta Qualifiers

The following subclauses list the meta qualifiers required for all CIM-compliant implementations. Meta qualifiers change the type of meta-element of the qualified schema element.

#### 5.6.2.1 Association

The Association qualifier takes boolean values, has Scope (Association) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the class is defining an association, i.e., its type of meta-element becomes Association.

#### 5.6.2.2 Indication

The Indication qualifier takes boolean values, has Scope (Class, Indication) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the class is defining an indication, i.e., its type of meta-element becomes Indication.

### 5.6.3 Standard Qualifiers

The following subclauses list the standard qualifiers required for all CIM-compliant implementations. Additional qualifiers can be supplied by extension classes to provide instances of the class and other operations on the class.

Not all of these qualifiers can be used together. The following principles apply:

- Not all qualifiers can be applied to all meta-model constructs. For each qualifier, the constructs to which it applies are listed.
- For a particular meta-model construct, such as associations, the use of the legal qualifiers may be further constrained because some qualifiers are mutually exclusive or the use of one qualifier implies restrictions on the value of another, and so on. These usage rules are documented in the subclause for each qualifier.
- Legal qualifiers are not inherited by meta-model constructs. For example, the MaxLen qualifier that applies to properties is not inherited by references.
- The meta-model constructs that can use a particular qualifier are identified for each qualifier. For qualifiers such as Association (see 5.6.2), there is an implied usage rule that the meta qualifier must also be present. For example, the implicit usage rule for the Aggregation qualifier (see 5.6.3.3) is that the Association qualifier must also be present.
- The allowed set of values for scope is (Class, Association, Indication, Property, Reference, Parameter, Method). Each qualifier has one or more of these scopes. If the scope is Class it does not apply to Association or Indication. If the scope is Property it does not apply to Reference.

#### 5.6.3.1 Abstract

The Abstract qualifier takes boolean values, has Scope (Class, Association, Indication) and has Flavor (Restricted). The default value is False.

This qualifier indicates that the class is abstract and serves only as a base for new classes. It is not possible to create instances of such classes.

#### 5.6.3.2 Aggregate

The Aggregate qualifier takes boolean values, has Scope (Reference) and has Flavor (DisableOverride). The default value is False.

The Aggregation and Aggregate qualifiers are used together. The Aggregation qualifier relates to the association, and the Aggregate qualifier specifies the parent reference.

#### 5.6.3.3 Aggregation

The Aggregation qualifier takes boolean values, has Scope (Association) and has Flavor (DisableOverride). The default value is False.

The Aggregation qualifier indicates that the association is an aggregation.

#### 5.6.3.4 ArrayType

The ArrayType qualifier takes string values, has Scope (Property, Parameter) and has Flavor (DisableOverride). The default value is "Bag".

The ArrayType qualifier is the type of the qualified array. Valid values are "Bag", "Indexed," and "Ordered."

For definitions of the array types, refer to 7.8.2.

The ArrayType qualifier shall be applied only to properties and method parameters that are arrays (defined using the square bracket syntax specified in ANNEX A).

The effective value of the ArrayType qualifier shall not change in the ancestry of the qualified element. This prevents incompatible changes in the behavior of the array element in subclasses.

NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied default value to an explicitly specified value.

#### 5.6.3.5 Bitmap

The Bitmap qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The Bitmap qualifier indicates the bit positions that are significant in a bitmap. The bitmap is evaluated from the right, starting with the least significant value. This value is referenced as 0 (zero). For example, using a uint8 data type, the bits take the form Mxxx xxxL, where M and L designate the most and least significant bits, respectively. The least significant bits are referenced as 0 (zero), and the most significant bit is 7. The position of a specific value in the Bitmap array defines an index used to select a string literal from the BitValues array.

The number of entries in the BitValues and Bitmap arrays shall match.

#### 5.6.3.6 BitValues

The BitValues qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The BitValues qualifier translates between a bit position value and an associated string. See 5.6.3.5 for the description for the Bitmap qualifier.

The number of entries in the BitValues and Bitmap arrays shall match.

#### 5.6.3.7 ClassConstraint

The ClassConstraint qualifier takes string array values, has Scope (Class, Association, Indication) and has Flavor (EnableOverride). The default value is Null.

The qualified element specifies one or more constraints that are defined in the OMG Object Constraint Language (OCL), as specified in the *Object Constraint Language* specification.

The ClassConstraint array contains string values that specify OCL definition and invariant constraints. The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of the qualified class, association, or indication.

OCL definition constraints define OCL attributes and OCL operations that are reusable by other OCL constraints in the same OCL context.

The attributes and operations in the OCL definition constraints shall be visible for:

- OCL definition and invariant constraints defined in subsequent entries in the same ClassConstraint array
- OCL constraints defined in PropertyConstraint qualifiers on properties and references in a class whose value (specified or inherited) of the ClassConstraint qualifier defines the OCL definition constraint
- Constraints defined in MethodConstraint qualifiers on methods defined in a class whose value (specified or inherited) of the ClassConstraint qualifier defines the OCL definition constraint

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

```
ocl_definition_string = "def" [ocl_name] ":" ocl_statement
```

Where:

`ocl_name` is the name of the OCL constraint.

`ocl_statement` is the OCL statement of the definition constraint, which defines the reusable attribute or operation.

An OCL invariant constraint is expressed as a typed OCL expression that specifies whether the constraint is satisfied. The type of the expression shall be boolean. The invariant constraint shall be satisfied at any time in the lifetime of the instance.

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

```
ocl_invariant_string = "inv" [ocl_name] ":" ocl_statement
```

Where:

`ocl_name` is the name of the OCL constraint.

`ocl_statement` is the OCL statement of the invariant constraint, which defines the boolean expression.

EXAMPLE 1: For example, to check that both property x and property y cannot be Null in any instance of a class, use the following qualifier, defined on the class:

```
ClassConstraint {
   "inv: not (self.x.oclIsUndefined() and self.y.oclIsUndefined())"
}
```

EXAMPLE 2: The same check can be performed by first defining OCL attributes. Also, the invariant constraint is named in the following example:

```
ClassConstraint {
   "def: xNull : Boolean = self.x.oclIsUndefined()",
   "def: yNull : Boolean = self.y.oclIsUndefined()",
   "inv xyNullCheck: xNull = False or yNull = False)"
}
```

#### 5.6.3.8 Composition

The Composition qualifier takes boolean values, has Scope (Association) and has Flavor (DisableOverride). The default value is False.

The Composition qualifier refines the definition of an aggregation association, adding the semantics of a whole-part/compositional relationship to distinguish it from a collection or basic aggregation. This refinement is necessary to map CIM associations more precisely into UML where whole-part relationships are considered compositions. The semantics conveyed by composition align with that of the *Unified Modeling Language: Superstructure*. Following is a quote (with emphasis added) from its section 7.3.3:

> "Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it."

Use of this qualifier imposes restrictions on the membership of the ‘collecting’ object (the whole). Care should be taken when entities are added to the aggregation, because they shall be "parts" of the whole. Also, if the collecting entity (the whole) is deleted, it is the responsibility of the implementation to dispose of the parts. The behavior may vary with the type of collecting entity whether the parts are also deleted. This is very different from that of a collection, because a collection may be removed without deleting the entities that are collected.

The Aggregation and Composition qualifiers are used together. Aggregation indicates the general nature of the association, and Composition indicates more specific semantics of whole-part relationships. This duplication of information is necessary because Composition is a more recent addition to the list of qualifiers. Applications can be built only on the basis of the earlier Aggregation qualifier.

#### 5.6.3.9 Correlatable

The Correlatable qualifier takes string array values, has Scope (Property) and has Flavor (EnableOverride). The default value is Null.

The Correlatable qualifier is used to define sets of properties that can be compared to determine if two CIM instances represent the same resource entity. For example, these instances may cross logical/physical boundaries, CIM server scopes, or implementation interfaces.

The sets of properties to be compared are defined by first specifying the organization in whose context the set exists (organization_name), and then a set name (set_name). In addition, a property is given a role name (role_name) to allow comparisons across the CIM Schema (that is, where property names may vary although the semantics are consistent).

The value of each entry in the Correlatable qualifier string array shall follow the formal syntax defined in ABNF:

```
correlatablePropertyID = organization_name ":" set_name ":" role_name
```

The determination whether two CIM instances represent the same resource entity is done by comparing one or more property values of each instance (where the properties are tagged by their role name), as follows: The property values of all role names within at least one matching organization name / set name pair shall match in order to conclude that the two instances represent the same resource entity. Otherwise, no conclusion can be reached and the instances may or may not represent the same resource entity.

`correlatablePropertyID` values shall be compared case-insensitively. For example,

```
"Acme:Set1:Role1" and "ACME:set1:role1"
```

are considered matching.

NOTE: The values of any string properties in CIM are defined to be compared case-sensitively.

To assure uniqueness of a `correlatablePropertyID`:

- organization_name shall include a copyrighted, trademarked or otherwise unique name that is owned by the business entity defining set_name, or is a registered ID that is assigned to the business entity by a recognized global authority. organization_name shall not contain a colon (":"). For DMTF defined `correlatablePropertyID` values, the organization_name shall be "CIM".
- set_name shall be unique within the context of organization_name and identifies a specific set of correlatable properties. set_name shall not contain a colon (":").
- role_name shall be unique within the context of organization_name and set_name and identifies the semantics or role that the property plays within the Correlatable comparison.

The Correlatable qualifier may be defined on only a single class. In this case, instances of only that class are compared. However, if the same correlation set (defined by organization_name and set_name) is specified on multiple classes, then comparisons can be done across those classes.

EXAMPLE: As an example, assume that instances of two classes can be compared: Class1 with properties PropA, PropB, and PropC, and Class2 with properties PropX, PropY and PropZ. There are two correlation sets defined, one set with two properties that have the role names Role1 and Role2, and the other set with one property with the role name OnlyRole. The following MOF represents this example:

```
Class1 {

     [Correlatable {"Acme:Set1:Role1"}]
   string PropA;

     [Correlatable {"Acme:Set2:OnlyRole"}]
   string PropB;

     [Correlatable {"Acme:Set1:Role2"}]
   string PropC;
};

Class2 {
```

```

      [Correlatable {"Acme:Set1:Role1"}]
    string PropX;

      [Correlatable {"Acme:Set2:OnlyRole"}]
    string PropY;

      [Correlatable {"Acme:Set1:Role2"}]
    string PropZ;
};
```

Following the comparison rules defined above, one can conclude that an instance of Class1 and an instance of Class2 represent the same resource entity if PropB and PropY's values match, or if PropA/PropX and PropC/PropZ's values match, respectively.

The Correlatable qualifier can be used to determine if multiple CIM instances represent the same underlying resource entity. Some may wonder if an instance's key value (such as InstanceID) is meant to perform the same role. This is not the case. InstanceID is merely an opaque identifier of a CIM instance, whereas Correlatable is not opaque and can be used to draw conclusions about the identity of the underlying resource entity of two or more instances.

DMTF-defined Correlatable qualifiers are defined in the CIM Schema on a case-by-case basis. There is no central document that defines them.

#### 5.6.3.10 Counter

The Counter qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is False.

The Counter qualifier applies only to unsigned integer types.

It represents a non-negative integer that monotonically increases until it reaches a maximum value of 2^n-1, when it wraps around and starts increasing again from zero. N can be 8, 16, 32, or 64 depending on the data type of the object to which the qualifier is applied. Counters have no defined initial value, so a single value of a counter generally has no information content.

#### 5.6.3.11 Deprecated

The Deprecated qualifier takes string array values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (Restricted). The default value is Null.

The Deprecated qualifier indicates that the CIM element (for example, a class or property) that the qualifier is applied to is considered deprecated. The qualifier may specify replacement elements. Existing CIM servers shall continue to support the deprecated element so that current CIM clients do not break. Existing CIM servers should add support for any replacement elements. A deprecated element should not be used in new CIM clients. Existing and new CIM clients shall tolerate the deprecated element and should move to any replacement elements as soon as possible. The deprecated element may be removed in a future major version release of the CIM schema, such as CIM 2.x to CIM 3.0.

The qualifier acts inclusively. Therefore, if a class is deprecated, all the properties, references, and methods in that class are also considered deprecated. However, no subclasses or associations or methods that reference that class are deprecated unless they are explicitly qualified as such. For clarity and to specify replacement elements, all such implicitly deprecated elements should be specifically qualified as deprecated.

The Deprecated qualifier's string value should specify one or more replacement elements. Replacement elements shall be specified using the following formal syntax defined in ABNF:

```
deprecatedEntry = className [ [ embeddedInstancePath ] "." elementSpec ]
```

where:

```
elementSpec = propertyName / methodName "(" [ parameterName *("," parameterName) ] ")"
```

is a specification of the replacement element.

```
embeddedInstancePath = 1*( "." propertyName )
```

is a specification of a path through embedded instances.

The qualifier is defined as a string array so that a single element can be replaced by multiple elements.

If there is no replacement element, then the qualifier string array shall contain a single entry with the string "No value".

When an element is deprecated, its description shall indicate why it is deprecated and how any replacement elements are used. Following is an acceptable example description:

"The X property is deprecated in lieu of the Y method defined in this class because the property actually causes a change of state and requires an input parameter."

The parameters of the replacement method may be omitted.

NOTE 1: Replacing a deprecated element with a new element results in duplicate representations of the element. This is of particular concern when deprecated classes are replaced by new classes and instances may be duplicated. To allow a CIM client to detect such duplication, implementations should document (in a ReadMe, MOF, or other documentation) how such duplicate instances are detected.

NOTE 2: Key properties may be deprecated, but they shall continue to be key properties and shall satisfy all rules for key properties. When a key property is no longer intended to be a key, only one option is available. It is necessary to deprecate the entire class and therefore its properties, methods, references, and so on, and to define a new class with the changed key structure.

#### 5.6.3.12 Description

The Description qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The Description qualifier describes a named element.

#### 5.6.3.13 DisplayName

The DisplayName qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The DisplayName qualifier defines a name that is displayed on a user interface instead of the actual name of the element.

#### 5.6.3.14 DN

The DN qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (DisableOverride). The default value is False.

When applied to a string element, the DN qualifier specifies that the string shall be a distinguished name as defined in Section 9 of ITU X.501 and the string representation defined in RFC2253. This qualifier shall not be applied to qualifiers that are not of the intrinsic data type string.

#### 5.6.3.15 EmbeddedInstance

The EmbeddedInstance qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

A non-Null effective value of this qualifier indicates that the qualified string typed element contains an embedded instance. The encoding of the instance contained in the string typed element qualified by EmbeddedInstance shall follow the rules defined in ANNEX F.

This qualifier may be used only on elements of string type.

If not Null the qualifier value shall specify the name of a CIM class in the same namespace as the class owning the qualified element. The embedded instance shall be an instance of the specified class, including instances of its subclasses.

The value of the EmbeddedInstance qualifier may be changed in subclasses to narrow the originally specified class to one of its subclasses. Other than that, the effective value of the EmbeddedInstance qualifier shall not change in the ancestry of the qualified element. This prevents incompatible changes between representing and not representing an embedded instance in subclasses.

See ANNEX F for examples.

#### 5.6.3.16 EmbeddedObject

The EmbeddedObject qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the qualified string typed element contains an encoding of an instance's data or an encoding of a class definition. The encoding of the object contained in the string typed element qualified by EmbeddedObject shall follow the rules defined in ANNEX F.

This qualifier may be used only on elements of string type.

The effective value of the EmbeddedObject qualifier shall not change in the ancestry of the qualified element. This prevents incompatible changes between representing and not representing an embedded object in subclasses.

NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied default value to an explicitly specified value.

See ANNEX F for examples.

#### 5.6.3.17 Exception

The Exception qualifier takes boolean values, has Scope (Indication) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the class and all subclasses of this class describe transient exception information. The definition of this qualifier is identical to that of the Abstract qualifier except that it cannot be overridden. It is not possible to create instances of exception classes.

The Exception qualifier denotes a class hierarchy that defines transient (very short-lived) exception objects. Instances of Exception classes communicate exception information between CIM entities. The Exception qualifier cannot be used with the Abstract qualifier. The subclass of an exception class shall be an exception class.

#### 5.6.3.18 Experimental

The Experimental qualifier takes boolean values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (Restricted). The default value is False.

If the Experimental qualifier is specified, the qualified element has experimental status. The implications of experimental status are specified by the schema owner.

In a DMTF-produced schema, experimental elements are subject to change and are not part of the final schema. In particular, the requirement to maintain backwards compatibility across minor schema versions does not apply to experimental elements. Experimental elements are published for developing implementation experience. Based on implementation experience, changes may occur to this element in future releases, it may be standardized "as is," or it may be removed. An implementation does not have to support an experimental feature to be compliant to a DMTF-published schema.

When applied to a class, the Experimental qualifier conveys experimental status to the class itself, as well as to all properties and features defined on that class. Therefore, if a class already bears the Experimental qualifier, it is unnecessary also to apply the Experimental qualifier to any of its properties or features, and such redundant use is discouraged.

No element shall be both experimental and deprecated (as with the Deprecated qualifier). Experimental elements whose use is considered undesirable should simply be removed from the schema.

#### 5.6.3.19 Gauge

The Gauge qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is False.

The Gauge qualifier is applicable only to unsigned integer types. It represents an integer that may increase or decrease in any order of magnitude.

The value of a gauge is capped at the implied limits of the property's data type. If the information being modeled exceeds an implied limit, the value represented is that limit. Values do not wrap. For unsigned integers, the limits are zero (0) to 2^n-1, inclusive. For signed integers, the limits are –(2^(n-1)) to 2^(n-1)-1, inclusive. N can be 8, 16, 32, or 64 depending on the data type of the property to which the qualifier is applied.

#### 5.6.3.20 In

The In qualifier takes boolean values, has Scope (Parameter) and has Flavor (DisableOverride). The default value is True.

This qualifier indicates that the qualified parameter is used to pass values to a method.

The effective value of the In qualifier shall not change in the ancestry of the qualified parameter. This prevents incompatible changes in the direction of parameters in subclasses.

NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied default value to an explicitly specified value.

#### 5.6.3.21 IsPUnit

The IsPUnit qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is False.

The qualified string typed property, method return value, or method parameter represents a programmatic unit of measure. The value of the string element follows the syntax for programmatic units.

The qualifier must be used on string data types only. A value of Null for the string element indicates that the programmatic unit is unknown. The syntax for programmatic units is defined in ANNEX C.

#### 5.6.3.22 Key

The Key qualifier takes boolean values, has Scope (Property, Reference) and has Flavor (DisableOverride). The default value is False.

The property or reference is part of the model path (see 8.2.5 for information on the model path). If more than one property or reference has the Key qualifier, then all such elements collectively form the key (a compound key).

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.

#### 5.6.3.23 MappingStrings

The MappingStrings qualifier takes string array values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

This qualifier indicates mapping strings for one or more management data providers or agents. See 5.6.6 for details.

#### 5.6.3.24 Max

The Max qualifier takes uint32 values, has Scope (Reference) and has Flavor (EnableOverride). The default value is Null.

The Max qualifier specifies the maximum cardinality of the reference, which is the maximum number of values a given reference may have for each set of other reference values in the association. For example, if an association relates A instances to B instances, and there shall be at most one A instance for each B instance, then the reference to A should have a Max(1) qualifier.

The Null value means that the maximum cardinality is unlimited.

#### 5.6.3.25 MaxLen

The MaxLen qualifier takes uint32 values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The MaxLen qualifier specifies the maximum length, in characters, of a string data item. MaxLen may be used only on string data types. If MaxLen is applied to CIM elements with a string array data type, it applies to every element of the array. A value of Null implies unlimited length.

An overriding property that specifies the MAXLEN qualifier must specify a maximum length no greater than the maximum length for the property being overridden.

#### 5.6.3.26 MaxValue

The MaxValue qualifier takes sint64 values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The MaxValue qualifier specifies the maximum value of this element. MaxValue may be used only on numeric data types. If MaxValue is applied to CIM elements with a numeric array data type, it applies to every element of the array. A value of Null means that the maximum value is the highest value for the data type.

An overriding property that specifies the MaxValue qualifier must specify a maximum value no greater than the maximum value of the property being overridden.

#### 5.6.3.27 MethodConstraint

The MethodConstraint qualifier takes string array values, has Scope (Method) and has Flavor (EnableOverride). The default value is Null.

The qualified element specifies one or more constraints, which are defined using the OMG Object Constraint Language (OCL), as specified in the *Object Constraint Language* specification.

The MethodConstraint array contains string values that specify OCL precondition, postcondition, and body constraints.

The OCL context of these constraints (that is, what "self" in OCL refers to) is the object on which the qualified method is invoked.

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 complete 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):

```
ocl_precondition_string = "pre" [ocl_name] ":" ocl_statement
```

Where:

`ocl_name` is the name of the OCL constraint.

`ocl_statement` is the OCL statement of the precondition constraint, which defines the boolean expression.

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):

```
ocl_postcondition_string = "post" [ocl_name] ":" ocl_statement
```

Where:

`ocl_name` is the name of the OCL constraint.

`ocl_statement` is the OCL statement of the post-condition constraint, which defines the boolean expression.

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 data type 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):

```
ocl_body_string = "body" [ocl_name] ":" ocl_statement
```

Where:

`ocl_name` is the name of the OCL constraint.

`ocl_statement` is the OCL statement of the body constraint, which defines the method return value.

EXAMPLE: The following qualifier defined on the RequestedStateChange( ) method of the CIM_EnabledLogicalElement class specifies that if a Job parameter is returned as not Null, then an CIM_OwningJobElement association must exist between the CIM_EnabledLogicalElement class and the Job.

```
MethodConstraint {
   "post AssociatedJob: "
      "not Job.oclIsUndefined() "
      "implies "
      "self.cIM_OwningJobElement.OwnedElement = Job"
}
```

### 5.6.3.28 Min

The Min qualifier takes uint32 values, has Scope (Reference) and has Flavor (EnableOverride). The default value is 0.

The Min qualifier specifies the minimum cardinality of the reference, which is the minimum number of values a given reference may have for each set of other reference values in the association. For example, if an association relates A instances to B instances and there shall be at least one A instance for each B instance, then the reference to A should have a Min(1) qualifier.

The qualifier value shall not be Null.

### 5.6.3.29 MinLen

The MinLen qualifier takes uint32 values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is 0.

The MinLen qualifier specifies the minimum length, in characters, of a string data item. MinLen may be used only on string data types. If MinLen is applied to CIM elements with a string array data type, it applies to every element of the array. The Null value is not allowed for MinLen.

An overriding property that specifies the MinLen qualifier must specify a minimum length no smaller than the minimum length of the property being overridden.

### 5.6.3.30 MinValue

The MinValue qualifier takes sint64 values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The MinValue qualifier specifies the minimum value of this element. MinValue may be used only on numeric data types. If MinValue is applied to CIM elements with a numeric array data type, it applies to every element of the array. A value of Null means that the minimum value is the lowest value for the data type.

An overriding property that specifies the MinValue qualifier must specify a minimum value no smaller than the minimum value of the property being overridden.

### 5.6.3.31 ModelCorrespondence

The ModelCorrespondence qualifier takes string array values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The ModelCorrespondence qualifier indicates a correspondence between two elements in the CIM schema. The referenced elements shall be defined in a standard or extension MOF file, such that the correspondence can be examined. If possible, forward referencing of elements should be avoided.

Object elements are identified using the following formal syntax defined in ABNF:

```
modelCorrespondenceEntry = className [ *( "." ( propertyName / referenceName ) )
                           [ "." methodName
                           [ "(" [ parameterName *( "," parameterName ) ] ")" ] ] ]
```

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 qualifiers may be used to describe a tighter coupling.

The following list provides examples of several correspondences found in CIM and vendor schemas:

- A vendor defines an Indication class corresponding to a particular CIM property or method so that Indications are generated based on the values or operation of the property or method. In this case, the ModelCorrespondence provides a correspondence between the property or method and the vendor's Indication class.
- 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. Also, referenced array properties may not be ordered but carry the default ArrayType qualifier definition of "Bag."
- 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 may define units, another property may define a multiplier, and another property may define a specific value. In this case, ModelCorrespondence is found on all related properties, each referencing all the others.
- Multi-dimensional arrays are desired. For example, one array may define names while another defines the name formats. In this case, the arrays are each defined with the ModelCorrespondence qualifier, referencing the other array properties or parameters. Also, they are indexed and they carry the ArrayType qualifier with the value "Indexed."

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

#### 5.6.3.32 NonLocal (removed)

This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0 of this document.

#### 5.6.3.33 NonLocalType (removed)

This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0 of this document.

#### 5.6.3.34 NullValue

The NullValue qualifier takes string values, has Scope (Property) and has Flavor (DisableOverride). The default value is Null.

The NullValue qualifier defines a value that indicates that the associated property is Null. That is, the property is considered to have a valid or meaningful value.

The NullValue qualifier may be used only with properties that have string and integer values. When used with an integer type, the qualifier value is a MOF decimal value as defined by the `decimalValue` ABNF rule defined in ANNEX A.

The content, maximum number of digits, and represented value are constrained by the data type of the qualified property.

This qualifier cannot be overridden because it seems unreasonable to permit a subclass to return a different Null value than that of the superclass.

#### 5.6.3.35 OctetString

The OctetString qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the qualified element is an octet string. An octet string is a sequence of octets and allows the representation of binary data.

The OctetString qualifier shall be specified only on elements of type array of uint8 or array of string.

When specified on elements of type array of uint8, the OctetString qualifier indicates that the entire array represents a single octet string. The first four array entries shall represent a length field, and any subsequent entries shall represent the octets in the octet string. The four uint8 values in the length field shall be interpreted as a 32-bit unsigned number where the first array entry is the most significant byte. The number represented by the length field shall be the number of octets in the octet string plus four. For example, the empty octet string is represented as { 0x00, 0x00, 0x00, 0x04 }.

When specified on elements of type array of string, the OctetString qualifier indicates that each array entry represents a separate octet string. The string value of each array entry shall be interpreted as a textual representation of the octet string. The string value of each array entry shall conform to the following formal syntax defined in ABNF:

```
"0x" 4*( hexDigit hexDigit )
```

The first four pairs of hexadecimal digits of the string value shall represent a length field, and any subsequent pairs shall represent the octets in the octet string. The four pairs of hexadecimal digits in the length field shall be interpreted as a 32-bit unsigned number where the first pair is the most significant byte. The number represented by the length field shall be the number of octets in the octet string plus four. For example, the empty octet string is represented as "0x00000004".

The effective value of the OctetString qualifier shall not change in the ancestry of the qualified element. This prevents incompatible changes in the interpretation of the qualified element in subclasses.

NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied default value to an explicitly specified value.

#### 5.6.3.36 Out

The Out qualifier takes boolean values, has Scope (Parameter) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the qualified parameter is used to return values from a method.

The effective value of the Out qualifier shall not change in the ancestry of the qualified parameter. This prevents incompatible changes in the direction of parameters in subclasses.

NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied default value to an explicitly specified value.

### 5.6.3.37 Override

The Override qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor (Restricted). The default value is Null.

If non-Null, the qualified element in the derived (containing) class takes the place of another element (of the same name) defined in the ancestry of that class.

The flavor of the qualifier is defined as 'Restricted' so that the Override qualifier is not repeated in (inherited by) each subclass. The effect of the override is inherited, but not the identification of the Override qualifier itself. This enables new Override qualifiers in subclasses to be easily located and applied.

An effective value of Null (the default) indicates that the element is not overriding any element. If not Null, the value shall conform to the following formal syntax defined in ABNF:

```
[ className"."]  IDENTIFIER
```

where `IDENTIFIER` shall be the name of the overridden element and if present, `className` shall be the name of a class in the ancestry of the derived class. The `className` ABNF rule shall be present if the class exposes more than one element with the same name (see 7.5.1).

If `className` is omitted, the overridden element is found by searching the ancestry of the class until a definition of an appropriately-named subordinate element (of the same meta-schema class) is found.

If `className` is specified, the element being overridden is found by searching the named class and its ancestry until a definition of an element of the same name (of the same meta-schema class) is found.

The Override qualifier may only refer to elements of the same meta-schema class. For example, properties can only override properties, etc. An element's name or signature shall not be changed when overriding.

### 5.6.3.38 Propagated

The Propagated qualifier takes string values, has Scope (Property) and has Flavor (DisableOverride). The default value is Null.

When the Propagated qualifier is specified with a non-Null value on a property, the Key qualifier shall be specified with a value of True on the qualified property.

A non-Null value of the Propagated qualifier indicates that the value of the qualified key property is propagated from a property in another instance that is associated via a weak association. That associated instance is referred to as the scoping instance of the instance receiving the property value.

A non-Null value of the Propagated qualifier shall identify the property in the scoping instance and shall conform to the formal syntax defined in ABNF:

```
[ className "." ] propertyName
```

where `propertyName` is the name of the property in the scoping instance, and `className` is the name of a class exposing that property. The specification of a class name may be needed in order to disambiguate like-named properties in associations with an arity of three or higher. It is recommended to specify the class name in any case.

For a description of the concepts of weak associations and key propagation as well as further rules around them, see 8.2

#### 5.6.3.39 PropertyConstraint

The PropertyConstraint qualifier takes string array values, has Scope (Property, Reference) and has Flavor (EnableOverride). The default value is Null.

The qualified element specifies one or more constraints that are defined using the Object Constraint Language (OCL) as specified in the *Object Constraint Language* specification.

The PropertyConstraint array contains string values that specify OCL initialization and derivation constraints. The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of the class, association, or indication that exposes the qualified property or reference.

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

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

```
ocl_initialization_string =  "init" ":" ocl_statement
```

Where:

`ocl_statement` is the OCL statement of the initialization constraint, which defines the typed expression.

An OCL derivation constraint is expressed as a typed OCL expression that specifies the permissible value for a property at any time in the lifetime of the instance. The type of the expression shall conform to the CIM data type of the property.

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

```
ocl_derivation_string = "derive" ":" ocl_statement
```

Where:

`ocl_statement` is the OCL statement of the derivation constraint, which defines the typed expression.

For example, PolicyAction has a SystemName property that must be set to the name of the system associated with CIM_PolicySetInSystem. The following qualifier defined on CIM_PolicyAction.SystemName specifies that constraint:

```
PropertyConstraint {
   "derive: self.CIM_PolicySetInSystem.Antecedent.Name"
}
```

A default value defined on a property also represents an initialization constraint, and no more than one initialization constraint is allowed on a property, as defined in 5.1.2.8.

No more than one derivation constraint is allowed on a property, as defined in 5.1.2.8.

#### 5.6.3.40 PUnit

The PUnit qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The PUnit qualifier indicates the programmatic unit of measure of the schema element. The qualifier value shall follow the syntax for programmatic units, as defined in ANNEX C.

The PUnit qualifier shall be specified only on schema elements of a numeric datatype. An effective value of Null indicates that a programmatic unit is unknown for or not applicable to the schema element.

String typed schema elements that are used to represent numeric values in a string format cannot have the PUnit qualifier specified, since 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 qualifier be set to True.

#### 5.6.3.41 Read

The Read qualifier takes boolean values, has Scope (Property) and has Flavor (EnableOverride). The default value is True.

The Read qualifier indicates that the property is readable.

#### 5.6.3.42 Reference

The Reference qualifier takes string values, has Scope (Property) and has Flavor (EnableOverride). The default value is NULL.

A non-NULL value of the Reference qualifier indicates that the qualified property references a CIM instance, and the qualifier value specifies the name of the class any referenced instance is of (including instances of subclasses of the specified class).

The value of a property with a non-NULL value of the Reference qualifier shall be the string representation of a CIM instance path (see 8.2.5) that references an instance of the class specified by the qualifier (including instances of subclasses of the specified class).

Note that the format of the string value representing the instance path depends on the usage context, as defined in 8.2.5. For example, when used in the context of a particular protocol, the property value is the string representation of instance paths defined for that protocol; when used in instance declarations in CIM MOF, the property value is the string representation of instance paths for CIM MOF, as defined in 8.5.

#### 5.6.3.43 Required

The Required qualifier takes boolean values, has Scope (Property, Reference, Parameter, Method) and has Flavor (DisableOverride). The default value is False.

A non-Null value is required for the element. For CIM elements with an array type, the Required qualifier affects the array itself, and the elements of the array may be Null regardless of the Required qualifier.

Properties of a class that are inherent characteristics of a class and identify that class are such properties as domain name, file name, burned-in device identifier, IP address, and so on. These properties are likely to be useful for CIM clients as query entry points that are not KEY properties but should be Required properties.

References of an association that are not KEY references shall be Required references. There are no particular usage rules for using the Required qualifier on parameters of a method outside of the meaning defined in this clause.

A property that overrides a required property shall not specify REQUIRED(False).

Compatible schema changes may add the Required qualifier to method output parameters, methods (i.e., their return values) and properties that may only be read. Compatible schema changes may remove the Required qualifier from method input parameters and properties that may only be written. If such compatible schema changes are done, the description of the changed schema element should indicate the schema version in which the change was made. This information can be used for example by management profile implementations in order to decide whether it is appropriate to implement a schema version higher than the one minimally required by the profile, and by CIM clients in order to decide whether they need to support both behaviors.

#### 5.6.3.44 Revision (deprecated)

**DEPRECATED**

The Revision qualifier is deprecated (See 5.6.3.55 for the description of the Version qualifier).

The Revision qualifier takes string values, has Scope (Class, Association, Indication) and has Flavor (EnableOverride, Translatable). The default value is Null.

The Revision qualifier provides the minor revision number of the schema object.

The Version qualifier shall be present to supply the major version number when the Revision qualifier is used.

**DEPRECATED**

#### 5.6.3.45 Schema (deprecated)

**DEPRECATED**

The Schema string qualifier is deprecated. The schema for any feature can be determined by examining the complete class name of the class defining that feature.

The Schema string qualifier takes string values, has Scope (Property, Method) and has Flavor (DisableOverride, Translatable). The default value is Null.

The Schema qualifier indicates the name of the schema that contains the feature.

**DEPRECATED**

#### 5.6.3.46 Source (removed)

This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0 of this document.

#### 5.6.3.47 SourceType (removed)

This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0 of this document.

#### 5.6.3.48 Static

The Static qualifier takes boolean values, has Scope (Property, Method) and has Flavor (DisableOverride). The default value is False.

The property or method is static. For a definition of static properties, see 7.5.5. For a definition of static methods, see 7.9.1.

An element that overrides a non-static element shall not be a static element.

#### 5.6.3.49 Structure

The Structure qualifier takes a boolean value, has Scope (Indication) and has Flavor (EnableOverride). The default value is False.

This qualifier indicates that the class describes a structure to be used as a property or parameter in a class along with the EmbeddedInstance. The definition of this qualifier is identical to that of the Abstract qualifier. It is not possible to create stand alone instances of structure classes.

The Structure qualifier denotes a class hierarchy that defines structure objects. The Structure qualifier cannot be used with the Abstract qualifier. The subclass of an structure class shall be a structure class.

#### 5.6.3.50 Terminal

The Terminal qualifier takes boolean values, has Scope (Class, Association, Indication) and has Flavor (EnableOverride). The default value is False.

The class can have no subclasses. If such a subclass is declared, the compiler generates an error.

This qualifier cannot coexist with the Abstract qualifier. If both are specified, the compiler generates an error.

#### 5.6.3.51 UMLPackagePath

The UMLPackagePath qualifier takes string values, has Scope (Class, Association, Indication) and has Flavor (EnableOverride). The default value is Null.

This qualifier specifies a position within a UML package hierarchy for a CIM class.

The qualifier value shall consist of a series of package names, each interpreted as a package within the preceding package, separated by '::'. The first package name in the qualifier value shall be the schema name of the qualified CIM class.

For example, consider a class named "CIM_Abc" that is in a package named "PackageB" that is in a package named "PackageA" that, in turn, is in a package named "CIM." The resulting qualifier specification for this class "CIM_Abc" is as follows:

```
UMLPACKAGEPATH ( "CIM::PackageA::PackageB" )
```

A value of Null indicates that the following default rule shall be used to create the UML package path: The name of the UML package path is the schema name of the class, followed by "::default".

For example, a class named "CIM_Xyz" with a UMLPackagePath qualifier value of Null has the UML package path "CIM::default".

#### 5.6.3.52 Units (deprecated)

---

**DEPRECATED**

The Units qualifier is deprecated.  Instead, the PUnit qualifier should be used for programmatic access, and the CIM client should use its own conventions to construct a string to be displayed from the PUnit qualifier.

The Units qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The Units qualifier specifies the unit of measure of the qualified property, method return value, or method parameter. For example, a Size property might have a unit of "Bytes."

Null indicates that the unit is unknown. An empty string indicates that the qualified property, method return value, or method parameter has no unit and therefore is dimensionless. The complete set of DMTF defined values for the Units qualifier is presented in ANNEX C.

**DEPRECATED**

#### 5.6.3.53 ValueMap

The ValueMap qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The ValueMap qualifier defines the set of permissible values for the qualified property, method return, or method parameter.

The ValueMap qualifier can be used alone or in combination with the Values qualifier. When it is used with the Values qualifier, the location of the value in the ValueMap array determines the location of the corresponding entry in the Values array.

ValueMap may be used only with string or integer types.

When used with a string typed element the following rules apply:

- a ValueMap entry shall be a string value as defined by the `stringValue` ABNF rule defined in ANNEX A.
- the set of ValueMap entries defined on a schema element may be extended in overriding schema elements in subclasses or in revisions of a schema within the same major version of the schema.

When used with an integer typed element the following rules apply:

- a ValueMap entry shall be a string value as defined by the `stringValue` ABNF rule defined in ANNEX A, whose string value conforms to the `integerValueMapEntry` ABNF rule:

```
integerValueMapEntry = integerValue / integerValueRange

integerValueRange = [integerValue] ".." [integerValue]
```

Where `integerValue` is defined in ANNEX A.

When used with an integer type, a ValueMap entry of:

`"x"` claims the value x.

`"..x"` claims all values less than and including x.

`"x.."` claims all values greater than and including x.

`".."` claims all values not otherwise claimed.

The values claimed are constrained by the value range of the data type of the qualified schema element.

The usage of `".."` as the only entry in the ValueMap array is not permitted.

If the ValueMap qualifier is used together with the Values qualifier, then all values claimed by a particular ValueMap entry apply to the corresponding Values entry.

EXAMPLE:

```
   [Values {"zero&one", "2to40", "fifty", "the unclaimed", "128-255"},
```

```
    ValueMap {"..1","2..40" "50", "..", "x80.."  }]
uint8 example;
```

In this example, where the type is uint8, the following mappings are made:

"`..1`" and "`zero&one`" map to 0 and 1.

"`2..40`" and "`2to40`" map to 2 through 40.

"`..`" and "`the unclaimed`" map to 41 through 49 and to 51 through 127.

"`0x80..`" and "`128-255`" map to 128 through 255.

An overriding property that specifies the ValueMap qualifier shall not map any values not allowed by the overridden property. In particular, if the overridden property specifies or inherits a ValueMap qualifier, then the overriding ValueMap qualifier must map only values that are allowed by the overridden ValueMap qualifier. However, the overriding property may organize these values differently than does the overridden property. For example, ValueMap {"0..10"} may be overridden by ValueMap {"0..1", "2..9"}. An overriding ValueMap qualifier may specify fewer values than the overridden property would otherwise allow.

#### 5.6.3.54 Values

The Values qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The Values qualifier translates between integer values and strings (such as abbreviations or English terms) in the ValueMap array, and an associated string at the same index in the Values array. If a ValueMap qualifier is not present, the Values array is indexed (zero relative) using the value in the associated property, method return type, or method parameter. If a ValueMap qualifier is present, the Values index is defined by the location of the property value in the ValueMap. If both Values and ValueMap are specified or inherited, the number of entries in the Values and ValueMap arrays shall match.

#### 5.6.3.55 Version

The Version qualifier takes string values, has Scope (Class, Association, Indication) and has Flavor (Restricted, Translatable). The default value is Null.

The Version qualifier provides the version information of the object, which increments when changes are made to the object.

Starting with CIM Schema 2.7 (including extension schema), the Version qualifier 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, M.N.U, as defined by the following ABNF:

```
versionFormat = decimalValue "." decimalValue "." decimalValue
```

The meaning of M.N.U is as follows:

**M** – The major version in numeric form of the change to the class.

**N** – The minor version in numeric form of the change to the class.

**U** – The update (for example, errata, patch, ...) in numeric form of the change to the class.

NOTE 1: The addition or removal of the Experimental qualifier does not require the version information to be updated.

NOTE 2: The 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.

EXAMPLES:

```
Version("2.7.0")

Version("1.0.0")
```

#### 5.6.3.56 Weak

The Weak qualifier takes boolean values, has Scope (Reference) and has Flavor (DisableOverride). The default value is False.

This qualifier indicates that the qualified reference is weak, rendering its owning association a weak association.

For a description of the concepts of weak associations and key propagation as well as further rules around them, see 8.2.

#### 5.6.3.57 Write

The Write qualifier takes boolean values, has Scope (Property) and has Flavor (EnableOverride). The default value is False.

The modeling semantics of a property support modification of that property by consumers. The purpose of this qualifier is to capture modeling semantics and not to address more dynamic characteristics such as provider capability or authorization rights.

#### 5.6.3.58 XMLNamespaceName

The XMLNamespaceName qualifier takes string values, has Scope (Property, Method, Parameter) and has Flavor (EnableOverride). The default value is Null.

The XMLNamespaceName qualifier shall be specified only on elements of type string or array of string.

If the effective value of the qualifier is not Null, this indicates that the value of the qualified element is an XML instance document. The value of the qualifier in this case shall be the namespace name of the XML schema to which the XML instance document conforms.

As defined in *Namespaces in XML*, the format of the namespace name shall be that of a URI reference as defined in RFC3986. Two such URI references may 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 a specification of the XMLNamespaceName qualifier overrides a non-Null qualifier value 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 19757-2:2008) is implied by usage of this qualifier.

### 5.6.4 Optional Qualifiers

The following subclauses list the optional qualifiers that address situations that are not common to all CIM-compliant implementations. Thus, CIM-compliant implementations can ignore optional qualifiers because they are not required to interpret or understand them. The optional qualifiers are provided in the specification to avoid random user-defined qualifiers for these recurring situations.

#### 5.6.4.1 Alias

The Alias qualifier takes string values, has Scope (Property, Reference, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The Alias qualifier establishes an alternate name for a property or method in the schema.

#### 5.6.4.2 Delete

The Delete qualifier takes boolean values, has Scope (Association, Reference) and has Flavor (EnableOverride). The default value is False.

**For associations**: The qualified association shall be deleted if any of the objects referenced in the association are deleted and the respective object referenced in the association is qualified with IfDeleted.

**For references**: The referenced object shall be deleted if the association containing the reference is deleted and qualified with IfDeleted. It shall also be deleted if any objects referenced in the association are deleted and the respective object referenced in the association is qualified with IfDeleted.

CIM clients shall chase associations according to the modeled semantic and delete objects appropriately.

NOTE: This usage rule must be verified when the CIM security model is defined.

#### 5.6.4.3 DisplayDescription

The DisplayDescription qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is Null.

The DisplayDescription qualifier defines descriptive text for the qualified element for display on a human interface — for example, fly-over Help or field Help.

The DisplayDescription qualifier is for use within extension subclasses of the CIM schema to provide display descriptions that conform to the information development standards of the implementing product. A value of Null indicates that no display description is provided. Therefore, a display description provided by the corresponding schema element of a superclass can be removed without substitution.

#### 5.6.4.4 Expensive

The Expensive qualifier takes boolean values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride).The default value is False.

The Expensive qualifier indicates that the element is expensive to manipulate and/or compute.

#### 5.6.4.5 IfDeleted

The IfDeleted qualifier takes boolean values, has Scope (Association, Reference) and has Flavor (EnableOverride). The default value is False.

All objects qualified by Delete within the association shall be deleted if the referenced object or the association, respectively, is deleted.

#### 5.6.4.6 Invisible

The Invisible qualifier takes boolean values, has Scope (Class, Association, Property, Reference, Method) and has Flavor (EnableOverride). The default value is False.

The Invisible qualifier indicates that the element is defined only for internal purposes and should not be displayed or otherwise relied upon. For example, an intermediate value in a calculation or a value to facilitate association semantics is defined only for internal purposes.

#### 5.6.4.7 Large

The Large qualifier takes boolean values, has Scope (Class, Property) and has Flavor (EnableOverride). The default value is False.

The Large qualifier property or class requires a large amount of storage space.

#### 5.6.4.8 PropertyUsage

The PropertyUsage qualifier takes string values, has Scope (Property) and has Flavor (EnableOverride). The default value is "CURRENTCONTEXT".

This qualifier allows properties to be classified according to how they are used by managed elements. Therefore, the managed element can convey intent for property usage. The qualifier does not convey what access CIM has to the properties. That is, not all configuration properties are writeable. Some configuration properties may be maintained by the provider or resource that the managed element represents, and not by CIM. The PropertyUsage qualifier enables the programmer to distinguish between properties that represent attributes of the following:

- A managed resource versus capabilities of a managed resource
- Configuration data for a managed resource versus metrics about or from a managed resource
- State information for a managed resource.

If the qualifier value is set to CurrentContext (the default value), then the value of PropertyUsage should be determined by looking at the class in which the property is placed. The rules for which default PropertyUsage values belong to which classes/subclasses are as follows:

Class>CurrentContext PropertyUsage Value
Setting > Configuration
Configuration > Configuration
Statistic > Metric ManagedSystemElement > State Product > Descriptive
FRU > Descriptive
SupportAccess > Descriptive
Collection > Descriptive

The valid values for this qualifier are as follows:

- **UNKNOWN.** The property's usage qualifier has not been determined and set.
- **OTHER.** The property's usage is not Descriptive, Capabilities, Configuration, Metric, or State.
- **CURRENTCONTEXT.** The PropertyUsage value shall be inferred based on the class placement of the property according to the following rules:
  – If the property is in a subclass of Setting or Configuration, then the PropertyUsage value of CURRENTCONTEXT should be treated as CONFIGURATION.
  – If the property is in a subclass of Statistics, then the PropertyUsage value of CURRENTCONTEXT should be treated as METRIC.
  – If the property is in a subclass of ManagedSystemElement, then the PropertyUsage value of CURRENTCONTEXT should be treated as STATE.

  – If the property is in a subclass of Product, FRU, SupportAccess or Collection, then the PropertyUsage value of CURRENTCONTEXT should be treated as DESCRIPTIVE.

- **DESCRIPTIVE.** The property contains information that describes the managed element, such as vendor, description, caption, and so on. These properties are generally not good candidates for representation in Settings subclasses.
- **CAPABILITY.** The property contains information that reflects the inherent capabilities of the managed element regardless of its configuration. These are usually specifications of a product. For example, VideoController.MaxMemorySupported=128 is a capability.
- **CONFIGURATION.** The property contains information that influences or reflects the configuration state of the managed element. These properties are candidates for representation in Settings subclasses. For example, VideoController.CurrentRefreshRate is a configuration value.
- **STATE** indicates that the property contains information that reflects or can be used to derive the current status of the managed element.
- **METRIC** indicates that the property contains a numerical value representing a statistic or metric that reports performance-oriented and/or accounting-oriented information for the managed element. This would be appropriate for properties containing counters such as "BytesProcessed".

#### 5.6.4.9 Provider

The Provider qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

An implementation-specific handle to a class implementation within a CIM server.

#### 5.6.4.10 Syntax

The Syntax qualifier takes string values, has Scope (Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The Syntax qualifier indicates the specific type assigned to a data item. It must be used with the SyntaxType qualifier.

#### 5.6.4.11 SyntaxType

The SyntaxType qualifier takes string values, has Scope (Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is Null.

The SyntaxType qualifier defines the format of the Syntax qualifier. It must be used with the Syntax qualifier.

#### 5.6.4.12 TriggerType

The TriggerType qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference, Method) and has Flavor (EnableOverride). The default value is Null.

The TriggerType qualifier specifies the circumstances that cause a trigger to be fired.

The trigger types vary by meta-model construct. For classes and associations, the legal values are CREATE, DELETE, UPDATE, and ACCESS. For properties and references, the legal values are UPDATE and ACCESS. For methods, the legal values are BEFORE and AFTER. For indications, the legal value is THROWN.

#### 5.6.4.13 UnknownValues

The UnknownValues qualifier takes string array values, has Scope (Property) and has Flavor (DisableOverride). The default value is Null.

The UnknownValues qualifier specifies a set of values that indicates that the value of the associated property is unknown. Therefore, the property cannot be considered to have a valid or meaningful value.

The conventions and restrictions for defining unknown values are the same as those for the ValueMap qualifier.

The UnknownValues qualifier cannot be overridden because it is unreasonable for a subclass to treat as known a value that a superclass treats as unknown.

#### 5.6.4.14 UnsupportedValues

The UnsupportedValues qualifier takes string array values, has Scope (Property) and has Flavor (DisableOverride). The default value is Null.

The UnsupportedValues qualifier specifies a set of values that indicates that the value of the associated property is unsupported. Therefore, the property cannot be considered to have a valid or meaningful value.

The conventions and restrictions for defining unsupported values are the same as those for the ValueMap qualifier.

The UnsupportedValues qualifier cannot be overridden because it is unreasonable for a subclass to treat as supported a value that a superclass treats as unknown.

### 5.6.5 User-defined Qualifiers

The user can define any additional arbitrary named qualifiers. However, it is recommended that only defined qualifiers be used and that the list of qualifiers be extended only if there is no other way to accomplish the objective.

### 5.6.6 Mapping Entities of Other Information Models to CIM

The MappingStrings qualifier 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 qualifier. 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 the defining bodies may define a part of the syntax used in the mapping.

#### 5.6.6.1 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. It 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" should 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 context of that protocol. This format is especially important for mapping variables defined in private MIBs. It 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 context 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 context 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

#### 5.6.6.2 General Mapping String Format

This clause defines the mapping string format, which provides a basis for future mapping string formats. Future mapping string formats defined in this document should be based on the general mapping string format. 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 8 is an example that defines a mapping string format based on the general mapping string format.

**Table 8 – Example for Mapping a String Format Based on the General Mapping String Format**

**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.

#### 5.6.6.3 MIF-Related Mapping String Format

Management Information Format (MIF) attributes can be mapped to CIM elements using the MappingStrings qualifier. This qualifier maps DMTF and vendor-defined MIF groups to CIM classes or properties using either domain or recast mapping.

---

**DEPRECATED**

MIF is defined in the DMTF *Desktop Management Interface Specification*, which completed DMTF end of life in 2005 and is therefore no longer considered relevant. Any occurrence of the MIF format in values of the MappingStrings qualifier is considered deprecated. Any other usage of MIF in this document is also considered deprecated. The MappingStrings qualifier itself is not deprecated because it is used for formats other than MIF.

**DEPRECATED**

---

As stated in the DMTF *Desktop Management Interface Specification*, every MIF group defines a unique identification that uses the MIF class string, which has the following formal syntax defined in ABNF:

```
mif_class_string = mif_defining_body "|" mif_specific_name "|" mif_version
```

Where:

```
mif_defining_body = 1*(stringChar)
```

is the name of the body defining the group. The dot ( . ) and vertical bar ( | ) characters are not allowed.

```
mif_specific_name = 1*(stringChar)
```

is the unique name of the group. The dot ( . ) and vertical bar ( | ) characters are not allowed.

```
mif_version = 3(decimalDigit)
```

is a three-digit number that identifies the version of the group definition.

The DMTF *Desktop Management Interface Specification* considers MIF class strings to be opaque identification strings for MIF groups. MIF class strings that differ only in whitespace characters are considered to be different identification strings.

In addition, each MIF attribute has a unique numeric identifier, starting with the number one, using the following formal syntax defined in ABNF:

```
mif_attribute_id = positiveDecimalDigit *decimalDigit
```

A MIF domain mapping maps an individual MIF attribute to a particular CIM property. A MIF recast mapping maps an entire MIF group to a particular CIM class.

The MIF format for use as a value of the MappingStrings qualifier has the following formal syntax defined in ABNF:

```
mif_format = mif_attribute_format | mif_group_format
```

Where:

```
mif_attribute_format = "MIF" "." mif_class_string "." mif_attribute_id
```

is used for mapping a MIF attribute to a CIM property.

```
mif_group_format = "MIF" "." mif_class_string
```

is used for mapping a MIF group to a CIM class.

For example, a MIF domain mapping of a MIF attribute to a CIM property is as follows:

```
   [MappingStrings { "MIF.DMTF|ComponentID|001.4" }]
string SerialNumber;
```

A MIF recast mapping maps an entire MIF group into a CIM class, as follows:

```
   [MappingStrings { "MIF.DMTF|Software Signature|002" }]
class SoftwareSignature
{
   ...
};
```

# 6 Managed Object Format

The management information is described in a language based on ISO/IEC 14750:1999 called the Managed Object Format (MOF). In this document, the term "MOF specification" refers to a collection of management information described in a way that conforms to the MOF syntax. Elements of MOF syntax are introduced on a case-by-case basis with examples. In addition, a complete description of the MOF syntax is provided in ANNEX A.

The MOF syntax describes object definitions in textual form and therefore establishes the syntax for writing definitions. The main components of a MOF specification are textual descriptions of classes, associations, properties, references, methods, and instance declarations and their associated qualifiers. Comments are permitted.

In addition to serving the need for specifying the managed objects, a MOF specification can be processed using a compiler. To assist the process of compilation, a MOF specification consists of a series of compiler directives.

MOF files shall be represented in Normalization Form C (NFC, defined in), and in one of the coded representation forms UTF-8, UTF-16BE or UTF-16LE (defined in ISO/IEC 10646:2003). UTF-8 is the recommended form for MOF files.

MOF files represented in UTF-8 should not have a signature sequence (EF BB BF, as defined in Annex H of ISO/IEC 10646:2003).

MOF files represented in UTF-16BE contain a big endian representation of the 16-bit data entities in the file; Likewise, MOF files represented in UTF-16LE contain little endian data entities. In both cases, they shall have a signature sequence (FEFF, as defined in Annex H of ISO/IEC 10646:2003).

Consumers of MOF files should use the signature sequence or absence thereof to determine the coded representation form.

This can be achieved by evaluating the first few Bytes in the file:

- FE FF → UTF-16BE
- FF FE → UTF-16LE
- EF BB BF → UTF-8
- otherwise → UTF-8

In order to test whether the 16-bit entities in the two UTF-16 cases need to be byte-wise swapped before processing, evaluate the first 16-bit data entity as a 16-bit unsigned integer. If it evaluates to 0xFEFF, there is no need to swap, otherwise (0xFFEF), there is a need to swap.

Consumers of MOF files shall ignore the UCS character the signature represents, if present.

## 6.1 MOF Usage

The managed object descriptions in a MOF specification can be validated against an active namespace (see clause 8). Such validation is typically implemented in an entity acting in the role of a CIM server. This clause describes the behavior of an implementation when introducing a MOF specification into a namespace. Typically, such a process validates both the syntactic correctness of a MOF specification and its semantic correctness against a particular implementation. In particular, MOF declarations must be ordered correctly with respect to the target implementation state. For example, if the specification references a class without first defining it, the reference is valid only if the CIM server already has a definition of that class. A MOF specification can be validated for the syntactic correctness alone, in a component such as a MOF compiler.

## 6.2 Class Declarations

A class declaration is treated as an instruction to create a new class. Whether the process of introducing a MOF specification into a namespace can add classes or modify classes is a local matter. If the specification references a class without first defining it, the CIM server must reject it as invalid if it does not already have a definition of that class.

## 6.3 Instance Declarations

Any instance declaration is treated as an instruction to create a new instance where the key values of the object do not already exist or an instruction to modify an existing instance where an object with identical key values already exists.

# 7 MOF Components

The following subclauses describe the components of MOF syntax.

## 7.1 Keywords

All keywords in the MOF syntax are case-insensitive.

## 7.2 Comments

Comments may appear anywhere in MOF syntax and are indicated by either a leading double slash ( // ) or a pair of matching /* and */ sequences.

A // comment is terminated by carriage return, line feed, or the end of the MOF specification (whichever comes first).

EXAMPLE:

```
// This is a comment
```

A /* comment is terminated by the next */ sequence or by the end of the MOF specification (whichever comes first). The meta model does not recognize comments, so they are not preserved across compilations. Therefore, the output of a MOF compilation is not required to include any comments.

## 7.3 Validation Context

Semantic validation of a MOF specification involves an explicit or implied namespace context. This is defined as the namespace against which the objects in the MOF specification are validated and the namespace in which they are created. Multiple namespaces typically indicate the presence of multiple management spaces or multiple devices.

## 7.4 Naming of Schema Elements

This clause describes the rules for naming schema elements, including classes, properties, qualifiers, methods, and namespaces.

CIM is a conceptual model that is not bound to a particular implementation. Therefore, it can be used to exchange management information in a variety of ways, examples of which are described in the Introduction clause. Some implementations may use case-sensitive technologies, while others may use case-insensitive technologies. The naming rules defined in this clause allow efficient implementation in either environment and enable the effective exchange of management information among all compliant implementations.

All names are case-insensitive, so two schema item names are identical if they differ only in case. This is mandated so that scripting technologies that are case-insensitive can leverage CIM technology. However, string values assigned to properties and qualifiers are not covered by this rule and must be treated as case-sensitive.

The case of a name is set by its defining occurrence and must be preserved by all implementations. This is mandated so that implementations can be built using case-sensitive technologies such as Java and object databases. This also allows names to be consistently displayed using the same user-friendly mixed-case format. For example, an implementation, if asked to create a Disk class must reject the request if there is already a DISK class in the current schema. Otherwise, when returning the name of the Disk class it must return the name in mixed case as it was originally specified.

CIM does not currently require support for any particular query language. It is assumed that implementations will specify which query languages are supported by the implementation and will adhere to the case conventions that prevail in the specified language. That is, if the query language is case-insensitive, statements in the language will behave in a case-insensitive way.

For the full rules for schema element names, see ANNEX A.

## 7.5 Class Declarations

A class is an object describing a grouping of data items that are conceptually related and that model an object. Class definitions provide a type system for instance construction.

### 7.5.1 Declaring a Class

A class is declared by specifying these components:

- Qualifiers of the class, which can be empty, or a list of qualifier name/value bindings separated by commas ( , ) and enclosed with square brackets ( [ and ] ).
- Class name.
- Name of the class from which this class is derived, if any.
- Class properties, which define the data members of the class. A property may also have an optional qualifier list expressed in the same way as the class qualifier list. In addition, a property has a data type, and (optionally) a default (initializer) value.
- Methods supported by the class. A method may have an optional qualifier list, and it has a signature consisting of its return type plus its parameters and their type and usage.
- A CIM class may expose more than one element (property or method) with a given name, but it is not permitted to define more than one element with a particular name. This can happen if a

base class defines an element with the same name as an element defined in a derived class without overriding the base class element. (Although considered rare, this could happen in a class defined in a vendor extension schema that defines a property or method that uses the same name that is later chosen by an addition to an ancestor class defined in the common schema.)

This sample shows how to declare a class:

```
   [abstract]
class Win32_LogicalDisk
{
      [read]
   string DriveLetter;

      [read, Units("KiloBytes")]
   sint32 RawCapacity = 0;

      [write]
   string VolumeLabel;

      [Dangerous]
   boolean Format([in] boolean FastFormat);
};
```

### 7.5.2 Subclasses

To indicate that a class is a subclass of another class, the derived class is declared by using a colon followed by the superclass name. For example, if the class ACME_Disk_v1 is derived from the class CIM_Media:

```
class ACME_Disk_v1 : CIM_Media
{
    // Body of class definition here ...
};
```

The terms base class, superclass, and supertype are used interchangeably, as are derived class, subclass, and subtype. The superclass declaration must appear at a prior point in the MOF specification or already be a registered class definition in the namespace in which the derived class is defined.

### 7.5.3 Default Property Values

Any properties (including references) in a class definition may have default values defined. The default value of a property represents an initialization constraint for the property and propagates to subclasses; for details see 5.1.2.8.

The format for the specification of a default value in CIM MOF depends on the property data type, and shall be:

- For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A.
- For the char16 datatype, as defined by the `charValue` or `integerValue` ABNF rules defined in ANNEX A.
- For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4. Since this is a string, it may be specified in multiple pieces, as defined by the `stringValue` ABNF rule defined in ANNEX A.

- For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.

For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.

- For <classname> REF datatypes, the string representation of the instance path as described in 8.5.

In addition, Null may be specified as a default value for any data type.

EXAMPLE:

```
class ACME_Disk
{
   string Manufacturer = "Acme";
   string ModelNumber  = "123-AAL";
};
```

As defined in 7.8.2, arrays can be defined to be of type Bag, Ordered, or Indexed. For any of these array types, a default value for the array may be specified by specifying the values of the array elements in a comma-separated list delimited with curly brackets, as defined in the `arrayInitializer` ABNF rule in ANNEX A.

EXAMPLE:

```
class ACME_ExampleClass
{
      [ArrayType ("Indexed")]
   string ip_addresses [] = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };
      // This variable length array has three elements as a default.

   sint32 sint32_values [10] = { 1, 2, 3, 5, 6 };
      // Since fixed arrays always have their defined number
      // of elements, default value defines a default value of Null
      // for the remaining elements.
};
```

### 7.5.4 Key Properties

Instances of a class can be identified within a namespace. Designating one or more properties with the Key qualifier provides for such instance identification. For example, this class has one property (Volume) that serves as its key:

```
class ACME_Drive
{
      [Key]
    string Volume;

    string FileSystem;

    sint32 Capacity;
};
```

The designation of a property as a key is inherited by subclasses of the class that specified the Key qualifier on the property. For example, the ACME_Modem class in the following example which subclasses the ACME_LogicalDevice class from the previous example, has the same two key properties as its superclass:

```
class ACME_Modem : ACME_LogicalDevice
{
   uint32 ActualSpeed;
};
```

A subclass that inherits key properties shall not designate additional properties as keys (by specifying the Key qualifier on them) and it shall not remove the designation as a key from any inherited key properties (by specifying the Key qualifier with a value of False on them).

Any non-abstract class shall expose key properties.

### 7.5.5 Static Properties

If a property is declared as a static property, it has the same value for all CIM instances that have the property in the same namespace. Therefore, any change in the value of a static property for a CIM instance also affects the value of that property for the other CIM instances that have it. As for any property, a change in the value of a static property of a CIM instance in one namespace may or may not affect its value in CIM instances in other namespaces.

Overrides on static properties are prohibited. Overrides of static methods are allowed.

## 7.6 Association Declarations

An association is a special kind of class describing a link between other classes. Associations also provide a type system for instance constructions. Associations are just like other classes with a few additional semantics, which are explained in the following subclauses.

### 7.6.1 Declaring an Association

An association is declared by specifying these components:

- Qualifiers of the association (at least the Association qualifier, if it does not have a supertype). Further qualifiers may be specified as a list of qualifier/name bindings separated by commas (,). The entire qualifier list is enclosed in square brackets ([ and ]).
- Association name. The name of the association from which this association derives (if any).
- Association references. Define pointers to other objects linked by this association. References may also have qualifier lists that are expressed in the same way as the association qualifier list — especially the qualifiers to specify cardinalities of references (see 5.1.2.14). In addition, a reference has a data type, and (optionally) a default (initializer) value.
- Additional association properties that define further data members of this association. They are defined in the same way as for ordinary classes.
- The methods supported by the association. They are defined in the same way as for ordinary classes.

EXAMPLE: The following example shows how to declare an association (assuming given classes CIM_A and CIM_B):

```
   [Association]
class CIM_LinkBetweenAandB : CIM_Dependency
{
```

```
      [Override ("Antecedent")]
   CIM_A REF Antecedent;

      [Override ("Dependent")]
   CIM_B REF Dependent;
};
```

### 7.6.2 Subassociations

To indicate a subassociation of another association, the same notation as for ordinary classes is used. The derived association is declared using a colon followed by the superassociation name. (An example is provided in 7.6.1).

### 7.6.3 Key References and Properties in Associations

Instances of an association class also can be identified within a namespace, because associations are just a special kind of a class. Designating one or more references or properties with the Key qualifier provides for such instance identification.

For example, this association class designates both of its references as keys:

```
   [Association, Aggregation]
class ACME_Component
{
      [Aggregate, Key]
   ACME_ManagedSystemElement REF GroupComponent;

      [Key]
   ACME_ManagedSystemElement REF PartComponent;
};
```

The key definition for associations follows the same rules as for ordinary classes: Compound keys are supported in the same way; keys are inherited by subassociations; Subassociations shall not add or remove keys.

These rules imply that associations may designate ordinary properties (i.e., properties that are not references) as keys and that associations may designate only a subset of its references as keys.

### 7.6.4 Weak Associations and Propagated Keys

CIM provides a mechanism to identify instances within the context of other associated instances. The class providing such context is called a *scoping class*, the class whose instances are identified within the context of the scoping class is called a *weak class*, and the association establishing the relation between these classes is called a *weak association*. Similarly, the instances of a scoping class are referred to as *scoping instances*, and the instances of a weak class are referred to as *weak instances*.

This mechanism allows weak instances to be identifiable in a global scope even though its own key properties do not provide such uniqueness on their own. The remaining keys come from the scoping class and provide the necessary context. These keys are called *propagated keys*, because they are propagated from the scoping instance to the weak instance.

An association is designated to be a weak association by qualifying the reference to the weak class with the Weak qualifier, as defined in 5.6.3.56. The propagated keys in the weak class are designated to be propagated by qualifying them with the Propagated qualifier, as defined in 5.6.3.38.

Figure 3 shows an example with two weak associations. There are three classes: ACME_ComputerSystem, ACME_OperatingSystem and ACME_LocalUser. ACME_OperatingSystem is weak with respect to ACME_ComputerSystem because the ACME_RunningOS association is marked as weak on its reference to ACME_OperatingSystem. Similarly, ACME_LocalUser is weak with respect to ACME_OperatingSystem because the ACME_HasUser association is marked as weak on its reference to ACME_LocalUser.

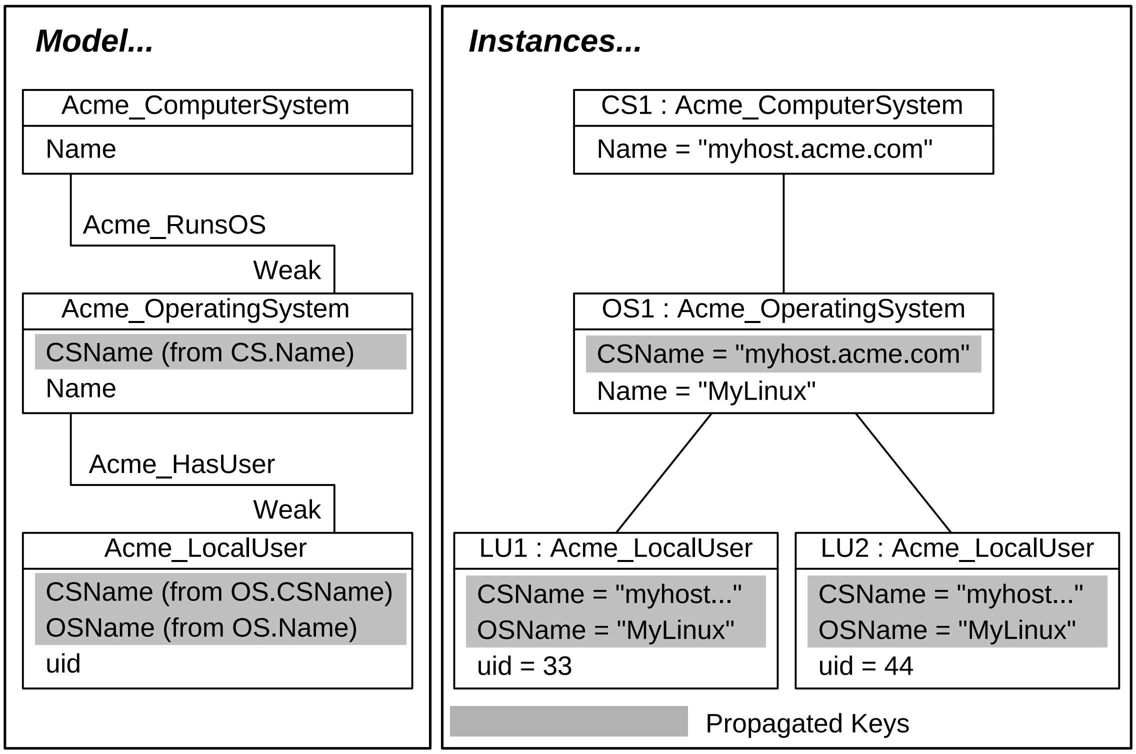



**Figure 3 – Example with Two Weak Associations and Propagated Keys**

The following MOF classes represent the example shown in Figure 3:

```
class ACME_ComputerSystem
{
      [Key]
   string Name;
};

class ACME_OperatingSystem
{
      [Key]
   string Name;

      [Key, Propagated ("ACME_ComputerSystem.Name")]
   string CSName;
};

class ACME_LocalUser
```

```
{
      [Key]
   String uid;

      [Key, Propagated("ACME_OperatingSystem.Name")]
   String OSName;

      [Key, Propagated("ACME_OperatingSystem.CSName")]
   String CSName;
};

   [Association]
class ACME_RunningOs
{
      [Key]
   ACME_ComputerSystem REF ComputerSystem;

      [Key, Weak]
   ACME_OperatingSystem REF OperatingSystem;
};

   [Association]
class ACME_HasUser
{
      [Key]
   ACME_OperatingSystem REF OperatingSystem;

      [Key, Weak]
   ACME_LocalUser REF User;
};
```

The following rules apply:

- A weak class may in turn be a scoping class for another class. In the example, ACME_OperatingSystem is scoped by ACME_ComputerSystem and scopes ACME_LocalUser.
- The property in the scoping instance that gets propagated does not need to be a key property.
- The association between the weak class and the scoping class shall expose a weak reference (see 5.6.3.56 "Weak") that targets the weak class.
- No more than one association may reference a weak class with a weak reference.
- An association may expose no more than one weak reference.
- Key properties may propagate across multiple weak associations. In the example, property Name in the ACME_ComputerSystem class is first propagated into class ACME_OperatingSystem as property CSName, and then from there into class ACME_LocalUser as property CSName (not changing its name this time). Still, only ACME_OperatingSystem is considered the scoping class for ACME_LocalUser.

NOTE: Since a reference to an instance always includes key values for the keys exposed by the class, a reference to an instance of a weak class includes the propagated keys of that class.

### 7.6.5 Object References

Object references are special properties whose values are links or pointers to other objects that are classes or instances. The value of an object reference is the string representation of an object path, as defined in 8.2. Consequently, the actual string value depends on the context the object reference is used in. For example, when used in the context of a particular protocol, the string value is the string representation defined for that protocol; when used in CIM MOF, the string value is the string representation of object paths for CIM MOF as defined in 8.5.

The data type of an object reference is declared as "XXX Ref", indicating a strongly typed reference to objects (instances or classes) of the class with name "XXX" or a subclass of this class. Object references in associations shall reference instances only and shall not have the special Null value.

---

**DEPRECATED**

Object references in method parameters shall reference instances or classes or both.

Note that only the use as relates to classes is deprecated.

**DEPRECATED**

---

Object references in method parameters shall reference instances.

Only associations may define references, ordinary classes and indications shall not define references, as defined in 5.1.2.13.

EXAMPLE 1:

```
   [Association]
class ACME_ExampleAssoc
{
   ACME_AnotherClass REF Inst1;
   ACME_Aclass       REF Inst2;
};
```

In this declaration, Inst1 can be set to point only to instances of type ACME_AnotherClass, including instances of its subclasses.

EXAMPLE 2:

```
class ACME_Modem
{
    uint32 UseSettingsOf (
        ACME_Modem REF OtherModem // references an instance object
    );
};
```

In this method, parameter OtherModem is used to reference an instance object.

The initialization of object references in association instances with object reference constants or aliases is defined in 7.8.

In associations, object references have cardinalities that are denoted using the Min and Max qualifiers. Examples of UML cardinality notations and their respective combinations of Min and Max values are shown in Table 9.

**Table 9 – UML Cardinality Notations**

| UML | MIN | MAX | Required MOF Text* | Description |
|---|---|---|---|---|
| * | 0 | Null | | Many |
| 1..* | 1 | Null | Min(1) | At least one |
| 1 | 1 | 1 | Min(1), Max(1) | One |
| 0,1 (or 0..1) | 0 | 1 | Max(1) | At most one |

## 7.7 Qualifiers

Qualifiers are named and typed values that provide information about CIM elements. Since the qualifier values are on CIM elements and not on CIM instances, they are considered to be meta-data.

This subclause describes how qualifiers are defined in MOF. For a description of the concept of qualifiers, see 5.6.1.

### 7.7.1 Qualifier Type

As defined in 5.6.1.2, the declaration of a qualifier type allows the definition of its name, data type, scope, flavor and default value.

The declaration of a qualifier type shall follow the formal syntax defined by the `qualifierDeclaration` ABNF rule defined in ANNEX A.

EXAMPLE 1:

The MaxLen qualifier which defines the maximum length of the string typed qualified element is declared as follows:

```
qualifier MaxLen : uint32 = Null,
   scope (Property, Method, Parameter);
```

This declaration establishes a qualifier named "MaxLen" that has a data type uint32 and can therefore specify length values between 0 and 2^32-1. It has scope (Property Method Parameter) and can therefore be specified on ordinary properties, method parameters and methods. It has no flavor specified, so it has the default flavor (ToSubclass EnableOverride) and therefore propagates to subclasses and is permitted to be overridden there. Its default value is NULL.

EXAMPLE 2:

The Deprecated qualifier which indicates that the qualified element is deprecated and allows the specification of replacement elements is declared as follows:

```
qualifier Deprecated : string[],
   scope (Any),
   flavor (Restricted);
```

This declaration establishes a qualifier named "Deprecated" that has a data type of array of string. It has scope (Any) and can therefore be defined on ordinary classes, associations, indications, ordinary properties, references, methods and method parameters. It has flavor (Restricted) and therefore does not propagate to subclasses. It has no default value defined, so its implied default value is Null.

### 7.7.2 Qualifier Value

As defined in 5.6.1.1, the specification of a qualifier defines a value for that qualifier on the qualified CIM element.

The specification of a set of qualifiers for a CIM element shall follow the formal syntax defined by the `qualifierList` ABNF rule defined in ANNEX A.

As defined there, specification of the `qualifierList` syntax element is optional, and if specified it shall be placed before the declaration of the CIM element the qualifiers apply to.

A specification of a qualifier in MOF requires that its qualifier type declaration be placed before the first specification of the qualifier on a CIM element.

EXAMPLE 1:

```
// Some qualifier type declarations

qualifier Abstract : boolean = False,
   scope (Class, Association, Indication),
   flavor (Restricted);

qualifier Description : string = Null,
   scope (Any),
   flavor (ToSubclass, EnableOverride, Translatable);

qualifier MaxLen : uint32 = Null,
   scope (Property, Method, Parameter),
   flavor (ToSubclass, EnableOverride);

qualifier ValueMap : string[],
   scope (Property, Method, Parameter),
   flavor (ToSubclass, EnableOverride);

qualifier Values : string[],
   scope (Property, Method, Parameter),
   flavor (ToSubclass, EnableOverride, Translatable);

// ...

// A class specifying these qualifiers

   [Abstract (True), Description (
      "A system.\n"
      "Details are defined in subclasses.")]
class ACME_System
{
      [MaxLen (80)]
   string Name;

      [ValueMap {"0", "1", "2", "3", "4..65535"},
       Values {"Not Applicable", "Unknown", "Other",
         "General Purpose", "Switch", "DMTF Reserved"}]
   uint16 Type;
};
```

In this example, the following qualifier values are specified:

- On class ACME_System:
  - A value of True for the Abstract qualifier
  - A value of "A system.\nDetails are defined in subclasses." for the Description qualifier
- On property Name:
  - A value of 80 for the MaxLen qualifier
- On property Type:
  - A specific array of values for the ValueMap qualifier
  - A specific array of values for the Values qualifier

As defined in 5.6.1.5, these CIM elements do have implied values for all qualifiers that are not specified but for which qualifier type declarations exist. Therefore, the following qualifier values are implied in addition in this example:

- On property Name:
  - A value of Null for the Description qualifier
  - An empty array for the ValueMap qualifier
  - An empty array for the Values qualifier
- On property Type:
  - A value of Null for the Description qualifier

Qualifiers may be specified without specifying a value. In this case, a default value is implied for the qualifier. The implied default value depends on the data type of the qualifier, as follows:

- For data type boolean, the implied default value is True
- For numeric data types, the implied default value is Null
- For string and char16 data types, the implied default value is Null
- For arrays of any data type, the implied default is that the array is empty.

EXAMPLE 2 (assuming the qualifier type declarations from example 1 in this subclause):

```
   [Abstract]
class ACME_Device
{
   // ...
};
```

In this example, the Abstract qualifier is specified without a value, therefore a value of True is implied on this boolean typed qualifier.

The concept of implying default values for qualifiers that are specified without a value is different from the concept of using the default values defined in the qualifier type declaration. The difference is that the latter is used when the qualifier is not specified. Consider the following example:

EXAMPLE 3 (assuming the declarations from examples 1 and 2 in this subclause):

```
class ACME_LogicalDevice : ACME_Device
{
   // ...
};
```

In this example, the Abstract qualifier is not specified, so its effective value is determined as defined in 5.6.1.5: Since the Abstract qualifier has flavor (Restricted), its effective value for class ACME_LogicalDevice is the default value defined in its qualifier type declaration, i.e., False, regardless of the value of True the Abstract qualifier has for class ACME_Device.

## 7.8 Instance Declarations

Instances are declared using the keyword sequence "instance of" and the class name. The property values of the instance may be initialized within an initialization block. Any qualifiers specified for the instance shall already be present in the defining class and shall have the same value and flavors.

Property initialization consists of an optional list of preceding qualifiers, the name of the property, and an optional value which defines the default value for the property as defined in 7.5.3. Any qualifiers specified for the property shall already be present in the property definition from the defining class, and they shall have the same value and flavors.

The format of initializer values for properties in instance declarations in CIM MOF depends on the data type of the property, and shall be:

- For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A.
- For the char16 datatype, as defined by the `charValue` or `integerValue` ABNF rules defined in ANNEX A.
- For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4. Since this is a string, it may be specified in multiple pieces, as defined by the `stringValue` ABNF rule defined in ANNEX A.
- For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.
- For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.
- For <classname> REF datatypes, as defined by the referenceInitializer ABNF rule defined in ANNEX A. This includes both object paths and instance aliases.

In addition, Null may be specified as an initializer value for any data type.

As defined in 7.8.2, arrays can be defined to be of type Bag, Ordered, or Indexed. For any of these array types, an array property can be initialized in an instance declaration by specifying the values of the array elements in a comma-separated list delimited with curly brackets, as defined in the `arrayInitializer` ABNF rule in ANNEX A.

For subclasses, all properties in the superclass may have their values initialized along with the properties in the subclass.

Any property values not explicitly initialized may be initialized by the implementation. If neither the instance declaration nor the implementation provides an intial value, a property is intialized to its default value if specified in the class definition. If still not initialized, the property is not assigned a value. The keyword NULL indicates the absence of value. The initial value of each property shall be conformant with any initialization constraints.

As defined in the description of the Key qualifier, the values of all key properties must be non-Null.

As described in item 21-E of subclause 5.1, a class may have, by inheritance, more than one property with a particular name. If a property initialization has a property name that applies to more than one property in the class, the initialization applies to the property defined closest to the class of the instance. That is, the property can be located by starting at the class of the instance. If the class defines a property with the name from the initialization, then that property is initialized. Otherwise, the search is repeated from the direct superclass of the class. See ANNEX H for more information about ambiguous property and method names.

For example, given the class definition:

```
class ACME_LogicalDisk : CIM_Partition
{
      [Key]
   string DriveLetter;

      [Units("kilo bytes")]
   sint32 RawCapacity = 128000;

      [Write]
   string VolumeLabel;

      [Units("kilo bytes")]
   sint32 FreeSpace;
};
```

an instance of this class can be declared as follows:

```
instance of ACME_LogicalDisk
{
    DriveLetter = "C";
    VolumeLabel = "myvol";
};
```

The resulting instance takes these property values:

- DriveLetter is assigned the value "C".
- RawCapacity is assigned the default value 128000.
- VolumeLabel is assigned the value "myvol".
- FreeSpace is assigned the value Null.

EXAMPLE: The following is an example with array properties:

```
class ACME_ExampleClass
{
      [ArrayType ("Indexed")]
   string ip_addresses [];    // Indexed array of variable length

   sint32 sint32_values [10]; // Bag array of fixed length = 10
};

instance of ACME_ExampleClass
{
   ip_addresses = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };
      // This variable length array now has three elements.

   sint32_values = { 1, 2, 3, 5, 6 };
      // Since fixed arrays always have their defined number
```

```
      // of elements, the remaining elements have the Null value.
};
```

EXAMPLE: The following is an example with instances of associations:

```
class ACME_Object
{
   string Name;
};

class ACME_Dependency
{
   ACME_Object REF Antecedent;
   ACME_Object REF Dependent;
};

instance of ACME_Dependency
{
   Dependent = "CIM_Object.Name = \"obj1\"";
   Antecedent = "CIM_Object.Name = \"obj2\"";
};
```

### 7.8.1 Instance Aliasing

Aliases are symbolic references to instances located elsewhere in the MOF specification. They have significance only within the MOF specification in which they are defined, and they are no longer available and have been resolved to instance paths once the MOF specification of instances has been loaded into a CIM server.

An alias can be assigned to an instance using the syntax defined for the `alias` ABNF rule in ANNEX A. Such an alias can later be used within the same MOF specification as a value for an object reference property.

Forward-referencing and circular aliases are permitted.

EXAMPLE:

```
class ACME_Node
{
   string Color;
};
```

These two instances define the aliases $Bluenode and $RedNode:

```
instance of ACME_Node as $BlueNode
{
   Color = "blue";
};

instance of ACME_Node as $RedNode
{
   Color = "red";
};
```

```
class ACME_Edge
{
   string Color;
   ACME_Node REF Node1;
   ACME_Node REF Node2;
};
```

These aliases $Bluenode and $RedNode are used in an association instance in order to reference the two instances.

```
instance of ACME_Edge
{
   Color = "green";
   Node1 = $BlueNode;
   Node2 = $RedNode;
};
```

### 7.8.2 Arrays

Arrays of any of the basic data types can be declared in the MOF specification by using square brackets after the property or parameter identifier. If there is an unsigned integer constant within the square brackets, the array is a fixed-length array and the constant indicates the size of the array; if there is nothing within the square brackets, the array is a variable-length array. Otherwise, the array definition is invalid.

Fixed-length arrays always have the specified number of elements. Elements cannot be added to or deleted from fixed-length arrays, but the values of elements can be changed.

Variable-length arrays have a number of elements between 0 and an implementation-defined maximum. Elements can be added to or deleted from variable-length array properties, and the values of existing elements can be changed.

Element addition, deletion, or modification is defined only for array properties because array parameters are only transiently instantiated when a CIM method is invoked. For array parameters, the array is thought to be created by the CIM client for input parameters and by the CIM server for output parameters. The array is thought to be retrieved and deleted by the CIM server for input parameters and by the CIM client for output parameters.

Array indexes start at 0 and have no gaps throughout the entire array, both for fixed-length and variable-length arrays. The special Null value signifies the absence of a value for an element, not the absence of the element itself. In other words, array elements that are Null exist in the array and have a value of Null. They do not represent gaps in the array.

The special Null value indicates that an array has no entries. That is, the set of entries of an empty array is the empty set. Thus if the array itself is equal to Null, then it is the empty array. This is distinguished from the case where the array is not equal to Null, but an entry of the array is equal to Null. The REQUIRED qualifier may be used to assert that an array shall not be Null.

The type of an array is defined by the ArraryType qualifier with values of Bag, Ordered, or Indexed. The default array type is Bag.

For a Bag array type, no significance is attached to the array index other than its convenience for accessing the elements of the array. There can be no assumption that the same index returns the same element for every retrieval, even if no element of the array is changed. The only valid assumption is that a retrieval of the entire array contains all of its elements and the index can be used to enumerate the complete set of elements within the retrieved array. The Bag array type should be used in the CIM schema when the order of elements in the array does not have a meaning. There is no concept of corresponding elements between Bag arrays.

For an Ordered array type, the CIM server maintains the order of elements in the array as long as no array elements are added, deleted, or changed. Therefore, the CIM server does not honor any order of elements presented by the CIM client when creating the array (during creation of the CIM instance for an array property or during CIM method invocation for an input array parameter) or when modifying the array. Instead, the CIM server itself determines the order of elements on these occasions and therefore possibly reorders the elements. The CIM server then maintains the order it has determined during successive retrievals of the array. However, as soon as any array elements are added, deleted, or changed, the CIM server again determines a new order and from then on maintains that new order. For output array parameters, the CIM server determines the order of elements and the CIM client sees the elements in that same order upon retrieval. The Ordered array type should be used when the order of elements in the array does have a meaning and should be controlled by the CIM server. The order the CIM server applies is implementation-defined unless the class defines particular ordering rules. Corresponding elements between Ordered arrays are those that are retrieved at the same index.

For an Indexed array type, the array maintains the reliability of indexes so that the same index returns the same element for successive retrievals. Therefore, particular semantics of elements at particular index positions can be defined. For example, in a status array property, the first array element might represent the major status and the following elements represent minor status modifications. Consequently, element addition and deletion is not supported for this array type. The Indexed array type should be used when the relative order of elements in the array has a meaning and should be controlled by the CIM client, and reliability of indexes is needed. Corresponding elements between Indexed arrays are those at the same index.

The current release of CIM does not support n-dimensional arrays.

Arrays of any basic data type are legal for properties. Arrays of references are not legal for properties. Arrays must be homogeneous; arrays of mixed types are not supported. In MOF, the data type of an array precedes the array name. Array size, if fixed-length, is declared within square brackets after the array name. For a variable-length array, empty square brackets follow the array name.

Arrays are declared using the following MOF syntax:

```
class ACME_A
{
      [Description("An indexed array of variable length"), ArrayType("Indexed")]
   uint8 MyIndexedArray[];

      [Description("A bag array of fixed length")]
   uint8 MyBagArray[17];
};
```

If default values are to be provided for the array elements, this MOF syntax is used:

```
class ACME_A
{
      [Description("A bag array property of fixed length")]
   uint8 MyBagArray[17] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};
};
```

EXAMPLE: The following MOF presents further examples of Bag, Ordered, and Indexed array declarations:

```
class ACME_Example
{
   char16 Prop1[];            // Bag (default) array of chars, Variable length
```

```

      [ArrayType ("Ordered")] // Ordered array of double-precision reals,
   real64 Prop2[];            // Variable length

      [ArrayType ("Bag")]     // Bag array containing 4 32-bit signed integers
   sint32 Prop3[4];

      [ArrayType ("Ordered")] // Ordered array of strings, Variable length
   string Prop4[] = {"an", "ordered", "list"};
      // Prop4 is variable length with default values defined at the
      // first three positions in the array

      [ArrayType ("Indexed")] // Indexed array of 64-bit unsigned integers
   uint64 Prop5[];
};
```

## 7.9 Method Declarations

A method is defined as an operation with a signature that consists of a possibly empty list of parameters and a return type. There are no restrictions on the type of parameters other than they shall be a fixed- or variable-length array of one of the data types described in 5.2. Method return types defined in MOF must be one of the data types described in 5.2. Return types cannot be arrays but are otherwise unrestricted.

Methods are expected, but not required, to return a status value indicating the result of executing the method. Methods may use their parameters to pass arrays.

Syntactically, the only thing that distinguishes a method from a property is the parameter list. The fact that methods are expected to have side-effects is outside the scope of this document.

EXAMPLE 1: In the following example, Start and Stop methods are defined on the CIM_Service class. Each method returns an integer value:

```
class CIM_Service : CIM_LogicalElement
{
      [Key]
   string Name;
   string StartMode;
   boolean Started;
   uint32 StartService();
   uint32 StopService();
};
```

EXAMPLE 2: In the following example, a Configure method is defined on the Physical DiskDrive class. It takes a DiskPartitionConfiguration object reference as a parameter and returns a boolean value:

```
class ACME_DiskDrive : CIM_Media
{
    sint32 BytesPerSector;
    sint32 Partitions;
    sint32 TracksPerCylinder;
    sint32 SectorsPerTrack;
    string TotalCylinders;
    string TotalTracks;
    string TotalSectors;
```

```
   string InterfaceType;
   boolean Configure([IN] DiskPartitionConfiguration REF config);
};
```

### 7.9.1 Static Methods

If a method is declared as a static method, it does not depend on any per-instance data. Non-static methods are invoked in the context of an instance; for static methods, the context of a class is sufficient. Overrides on static properties are prohibited. Overrides of static methods are allowed.

## 7.10 Compiler Directives

Compiler directives are provided as the keyword "pragma" preceded by a hash ( # ) character and followed by a string parameter. The current standard compiler directives are listed in Table 10.

**Table 10 – Standard Compiler Directives**

| Compiler Directive | Interpretation |
|---|---|
| #pragma include() | Has a file name as a parameter. The file is assumed to be a MOF file. The pragma has the effect of textually inserting the contents of the include file at the point where the include pragma is encountered. |
| #pragma instancelocale() | Declares the locale used for instances described in a MOF file. This pragma specifies the locale when "INSTANCE OF" MOF statements include string or char16 properties and the locale is not the same as the locale specified by a #pragma locale() statement. The locale is specified as a parameter of the form ll_cc where ll is a language code as defined in ISO 639-1:2002, ISO649-2:1999, or ISO 639-3:2007 and cc is a country code as defined in ISO 3166-1:2006, ISO 3166-2:2007, or ISO 3166-3:1999. |
| #pragma locale() | Declares the locale used for a particular MOF file. The locale is specified as a parameter of the form ll_cc, where ll is a language code as defined in ISO 639-1:2002, ISO649-2:1999, or ISO 639-3:2007 and cc is a country code as defined in ISO 3166-1:2006, ISO 3166-2:2007, or ISO 3166-3:1999. When the pragma is not specified, the assumed locale is "en_US".<br>This pragma does not apply to the syntax structures of MOF. Keywords, such as "class" and "instance", are always in en_US. |
| #pragma namespace() | This pragma is used to specify a Namespace path. |
| #pragma nonlocal() | These compiler directives and the corresponding instance-level qualifiers were removed as an erratum in version 2.3.0 of this document. |
| #pragma nonlocaltype() | |
| #pragma source() | |
| #pragma sourcetype() | |

Pragma directives may be added as a MOF extension mechanism. Unless standardized in a future CIM infrastructure specification, such new pragma definitions must be considered vendor-specific. Use of non-standard pragma affects the interoperability of MOF import and export functions.

## 7.11 Value Constants

The constant types supported in the MOF syntax are described in the subclauses that follow. These are used in initializers for classes and instances and in the parameters to named qualifiers.

For a formal specification of the representation, see ANNEX A.

### 7.11.1 String Constants

A string constant in MOF is represented as a sequence of one or more string constant parts, separated by whitespace or comments. Each string constant part is enclosed in double-quotes (") and contains zero or more UCS characters or escape sequences. Double quotes shall be escaped. The character repertoire for these UCS characters is defined in 5.2.2.

The following escape sequences are defined for string constants:

\b // U+0008: backspace

\t // U+0009: horizontal tab

\n // U+000A: linefeed

\f // U+000C: form feed

\r // U+000D: carriage return

\" // U+0022: double quote (")

\' // U+0027: single quote (')

\\ // U+005C: backslash (\)

\x<hex> // a UCS character, where <hex> is one to four hex digits, representing its UCS code position

\X<hex> // a UCS character, where <hex> is one to four hex digits, representing its UCS code position

The \x<hex> and \X<hex> forms are limited to represent only the UCS-2 character set.

For example, the following is a valid string constant:

```
"This is a string"
```

Successive quoted strings are concatenated as long as only whitespace or a comment intervenes:

```
"This"  " becomes a long string"
"This" /* comment */ " becomes a long string"
```

### 7.11.2 Character Constants

A character constant in MOF is represented as one UCS character or escape sequence enclosed in single quotes ('), or as an integer constant as defined in 7.11.3. The character repertoire for the UCS character is defined in 5.2.3. The valid escape sequences are defined in 7.11.1. Single quotes shall be escaped. An integer constant represents the code position of a UCS character and its character repertoire is defined in 5.2.3.

For example, the following are valid character constants:

```
'a'         // U+0061: 'a'
'\n'        // U+000A: linefeed
'1'         // U+0031: '1'
'\x32'      // U+0032: '2'
65          // U+0041: 'A'
0x41        // U+0041: 'A'
```

### 7.11.3 Integer Constants

Integer constants may be decimal, binary, octal, or hexadecimal. For example, the following constants are all legal:

```
1000
-12310
0x100
01236
100101B
```

Binary constants have a series of 1 and 0 digits, with a "b" or "B" suffix to indicate that the value is binary.

The number of digits permitted depends on the current type of the expression. For example, it is not legal to assign the constant 0xFFFF to a property of type uint8.

### 7.11.4 Floating-Point Constants

Floating-point constants are declared as specified by ANSI/IEEE 754-1985. For example, the following constants are legal:

```
3.14
-3.14
-1.2778E+02
```

The range for floating-point constants depends on whether float or double properties are used, and they must fit within the range specified for 4-byte and 8-byte floating-point values, respectively.

### 7.11.5 Object Reference Constants

As defined in 7.6.5, object references are special properties whose values are links or pointers to other objects, which may be classes or instances. Object reference constants are string representations of object paths for CIM MOF, as defined in 8.5.

The usage of object reference constants as initializers for instance declarations is defined in 7.8, and as default values for properties in 7.5.3.

### 7.11.6 Null

The predefined constant NULL represents the absence of value. See 5.2 for details

.

## 8 Naming

Because CIM is not bound to a particular technology or implementation, it promises to facilitate sharing management information among a variety of management platforms. The CIM naming mechanism addresses the following requirements:

- Ability to unambiguously reference CIM objects residing in a CIM server.
- Ability for CIM object names to be represented in multiple protocols, and for these representations the ability to be transformed across such protocols in an efficient manner.
- Support the following types of CIM objects to be referenced: instances, classes, qualifier types and namespaces.

- Ability to determine when two object names reference the same CIM object. This entails location transparency so that there is no need for a consumer of an object name to understand which management platforms proxy the instrumentation of other platforms.

The Key qualifier is the CIM Meta-Model mechanism to identify the properties that uniquely identify an instance of a class (including an instance of an association) within a CIM namespace. This clause defines how CIM instances, classes, qualifier types and namespaces are referenced using the concept of CIM object paths.

## 8.1 CIM Namespaces

Because CIM allows multiple implementations, it is not sufficient to think of the name of a CIM instance as just the combination of its key properties. The instance name must also identify the implementation that actually hosts the instances. In order to separate the concept of a run-time container for CIM objects represented by a CIM server from the concept of naming, CIM defines the notion of a CIM namespace. This separation of concepts allows separating the design of a model along the boundaries of namespaces from the placement of namespaces in CIM servers.

A namespace provides a scope of uniqueness for some types of object. Specifically, the names of class objects and of qualifier type objects shall be unique in a namespace. The compound key of instance objects shall be unique across all instances of the class (not including subclasses) within the namespace.

In addition, a namespace is considered a CIM object since it is addressable using an object name. However, a namespace cannot host other namespaces, in other words the set of namespaces in a CIM server is flat. A namespace has a name which shall be unique within the CIM server.

A namespace is also considered a run-time container within a CIM server which can host objects. For example, CIM objects are said to reside in namespaces as well as in CIM servers. Also, a common notion is to load the definition of qualifier types, classes and instances into a namespace, where they become objects that can be referenced. The run-time aspect of a CIM namespace makes it different from other definitions of namespace concepts that are addressing only the name uniqueness aspect, such as namespaces in Java, C++ or XML.

## 8.2 Naming CIM Objects

This subclause defines a concept for naming the objects residing in a CIM server. The naming concept allows for unambiguously referencing these objects and supports the following types of objects:

- namespaces
- qualifier types
- classes
- instances

### 8.2.1 Object Paths

The construct that references an object residing in a CIM server is called an object path. Since CIM is independent of implementations and protocols, object paths are defined in an abstract way that allows for defining different representations of the object paths. Protocols using object paths are expected to define representations of object paths as detailed in this subclause. A representation of object paths for CIM MOF is defined in 8.5.

**DEPRECATED**

Before version 2.6.0 of this document, object paths were referred to as "object names". The term "object name" is deprecated since version 2.6.0 of this document and the term "object path" should be used instead.

**DEPRECATED**

An object path is defined as a hierarchy of naming components. The leaf components in that hierarchy have a string value that is defined in this document. It is up to specifications using object paths to define how the string values of the leaf components are assembled into their own string representation of an object path, as defined in 8.4.

Figure 4 shows the general hierarchy of naming components of an object path. The naming components are defined more specifically for each type of object supported by CIM naming. The leaf components are shown with gray background.

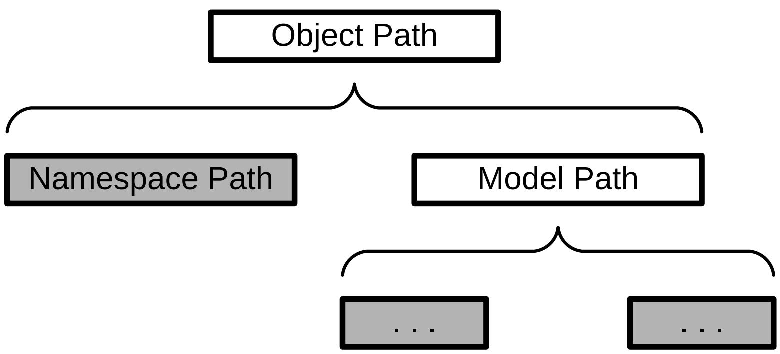



**Figure 4 – General Component Structure of Object Path**

Generally, an object path consists of two naming components:

- namespace path – an unambiguous reference to the namespace in a CIM server, and
- model path – an unambiguous identification of the object relative to that namespace.

This document does not define the internal structure of a namespace path, but it defines requirements on specifications using object paths in 8.4, including a requirement for a string representation of the namespace path.

A model path can be described using CIM model elements only. Therefore, this document defines the naming components of the model path for each type of object supported by CIM naming. Since the leaf components of model paths are CIM model elements, their string representation is well defined and specifications using object paths only need to define how these strings are assembled into an object path, as defined in 8.4.

The definition of a string representation for object paths is left to specifications using object paths, as described in 8.4.

Two object paths match if their namespace path components match, and their model path components (if any) have matching leaf components. As a result, two object paths that match reference the same CIM object.

NOTE: The matching of object paths is not just a simple string comparison of the string representations of object paths.

### 8.2.2 Object Path for Namespace Objects

The object path for namespace objects is called namespace path. It consists of only the Namespace Path component, as shown in Figure 5. A Model Path component is not present.

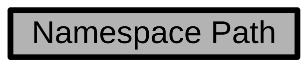



**Figure 5 – Component Structure of Object Path for Namespaces**

The definition of a string representation for namespace paths is left to specifications using object paths, as described in 8.4.

Two namespace paths match if they reference the same namespace. The definition of a method for determining whether two namespace paths reference the same namespace is left to specifications using object paths, as described in 8.4.

The resulting method may or may not be able to determine whether two namespace paths reference the same namespace. For example, there may be alias names for namespaces, or different ports exposing access to the same namespace. Often, specifications using object paths need to revert to the minimally possible conclusion which is that namespace paths with equal string representations reference the same namespace, and that for namespace paths with unequal string representations no conclusion can be made about whether or not they reference the same namespace.

### 8.2.3 Object Path for Qualifier Type Objects

The object path for qualifier type objects is called qualifier type path. Its naming components have the structure defined in Figure 6.

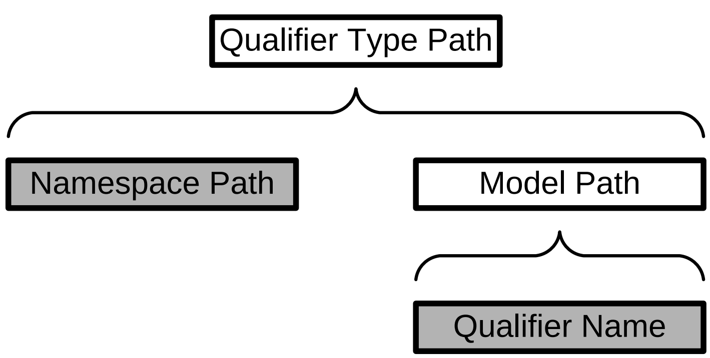



**Figure 6 – Component Structure of Object Path for Qualifier Types**

The Namespace Path component is defined in 8.2.2.

The string representation of the Qualifier Name component shall be the name of the qualifier, preserving the case defined in the namespace. For example, the string representation of the Qualifier Name component for the MappingStrings qualifier is "MappingStrings".

Two Qualifier Names match as described in 8.2.6.

### 8.2.4 Object Path for Class Objects

The object path for class objects is called class path. Its naming components have the structure defined in Figure 7.

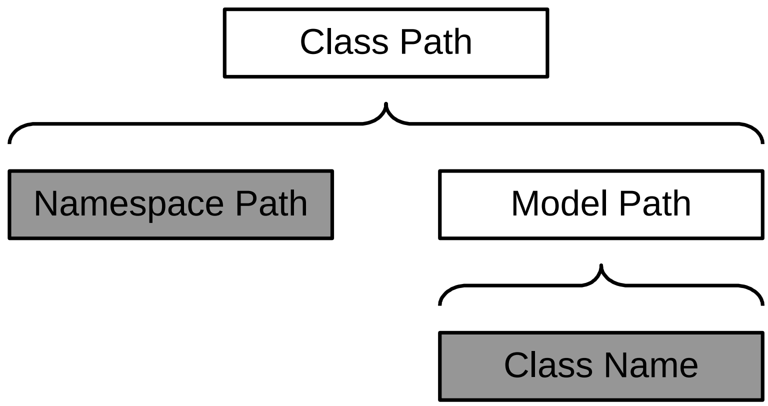



**Figure 7 – Component Structure of Object Path for Classes**

The Namespace Path component is defined in 8.2.2.

The string representation of the Qualifier Name component shall be the name of the qualifier, preserving the case defined in the namespace. For example, the string representation of the Qualifier Name component for the MappingStrings qualifier is "MappingStrings".

Two Qualifier Names match as described in 8.2.6.

### 8.2.5 Object Path for Class Objects

The object path for instance objects is called ***instance path***. Its naming components have the structure defined in Figure 7.

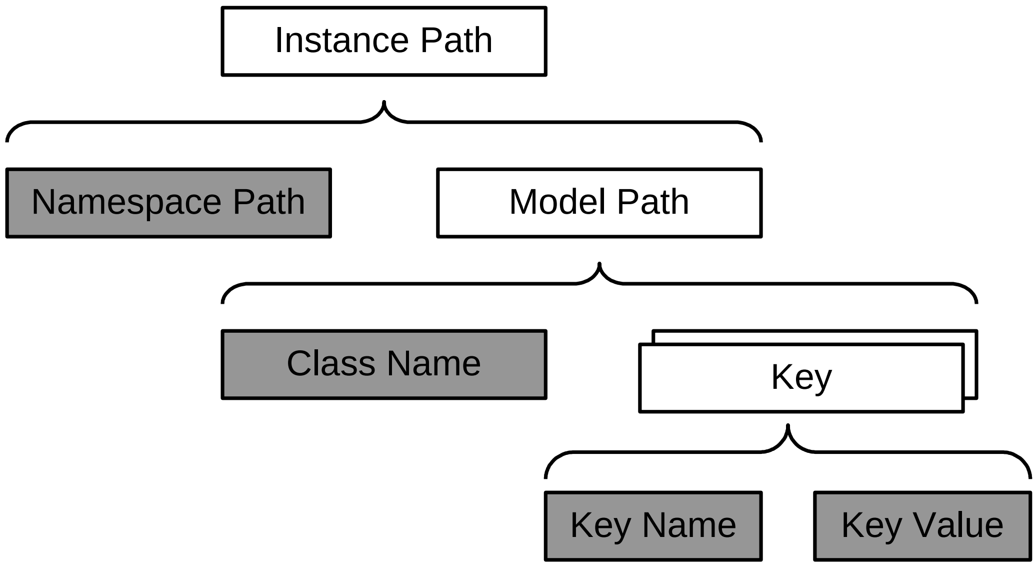



**Figure 8 – Component Structure of Object Path for Instances**

The Namespace Path component is defined in 8.2.2.

The Class Name component is defined in 8.2.4.

The Model Path component consists of a Class Name component and an unordered set of one or more Key components. There shall be one Key component for each key property (including references) exposed by the class of the instance. The set of key properties includes any propagated keys, as defined in 7.6.4. There shall not be Key components for properties (including references) that are not keys. Classes that do not expose any keys cannot have instances that are addressable with an object path for instances.

The string representation of the Key Name component shall be the name of the key property, preserving the case defined in the class residing in the namespace. For example, the string representation of the Key Name component for a property ActualSpeed defined in a class ACME_Device is "ActualSpeed".

Two Key Names match as described in 8.2.6.

The Key Value component represents the value of the key property. The string representation of the Key Value component is defined by specifications using object names, as defined in 8.4.

Two Key Values match as defined for the datatype of the key property.

### 8.2.6 Matching CIM Names

Matching of CIM names (which consist of UCS characters) as defined in this document shall be performed as if the following algorithm was applied:

Any lower case UCS characters in the CIM names are translated to upper case.

The CIM names are considered to match if the string identity matching rules defined in chapter 4 "String Identity Matching" of *Character Model for the World Wide Web 1.0: Normalization* match when applied to the upper case CIM names.

In order to eliminate the costly processing involved in this, specifications using object paths may define simplified processing for applying this algorithm. One way to achieve this is to mandate that Normalization Form C (NFC), defined in *The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms*, which allows the normalization to be skipped when comparing the names.

## 8.3 Identity of CIM Objects

As defined in 8.2.1, two CIM objects are identical if their object paths match. Since this depends on whether their namespace paths match, it may not be possible to determine this (for details, see 8.2.2).

Two different CIM objects (e.g., instances) can still represent aspects of the same managed object. In other words, identity at the level of CIM objects is separate from identity at the level of the represented managed objects.

## 8.4 Requirements on Specifications Using Object Paths

This subclause comprehensively defines the CIM naming related requirements on specifications using CIM object paths:

> Such specifications shall define a string representation of a namespace path (referred to as "namespace path string") using an ABNF syntax that defines its specification dependent components. The ABNF syntax shall not have any ABNF rules that are considered opaque or undefined. The ABNF syntax shall contain an ABNF rule for the namespace name.

A namespace path string as defined with that ABNF syntax shall be able to reference a namespace object in a way that is unambiguous in the environment where the CIM server hosting the namespace is expected to be used. This typically translates to enterprise wide addressing using Internet Protocol addresses.

Such specifications shall define a method for determining from the namespace path string the particular object path representation defined by the specification. This method should be based on the ABNF syntax defined for the namespace path string.

Such specifications shall define a method for determining whether two namespace path strings reference the same namespace. As described in 8.2.2, this method may not support this in any case.

Such specifications shall define how a string representation of the object paths for qualifier types, classes and instances is assembled from the string representations of the leaf components defined in 8.2.1 to 8.2.5, using an ABNF syntax.

Such specifications shall define string representations for all CIM datatypes that can be used as keys, using an ABNF syntax.

## 8.5 Object Paths Used in CIM MOF

Object paths are used in CIM MOF to reference instance objects in the following situations:

- when specifying default values for references in association classes, as defined in 7.5.3.
- when specifying initial values for references in association instances, as defined in 7.8.

In CIM MOF, object paths are not used to reference namespace objects, class objects or qualifier type objects.

The string representation of instance paths used in CIM MOF shall conform to the `WBEM-URI-UntypedInstancePath` ABNF rule defined in subclause 4.5 "Collected BNF for WBEM URI" of DSP0207.

That subclause also defines:

- a string representation for the namespace path.
- how a string representation of an instance path is assembled from the string representations of the leaf components defined in 8.2.1 to 8.2.5.
- how the namespace name is determined from the string representation of an instance path.

That specification does not presently define a method for determining whether two namespace path strings reference the same namespace.

The string representations for key values shall be:

- For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A, as one single string.
- For the char16 datatype, as defined by the `charValue` ABNF rule defined in ANNEX A.
- For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4, as one single string.
- For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.
- For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.
- For <classname> REF datatypes, the string representation of the instance path as described in this subclause.

EXAMPLE: Examples for string representations of instance paths in CIM MOF are as follows:

```
"http://myserver.acme.com/root/cimv2:ACME_LogicalDisk.SystemName=\"acme\",Drive=\"C\""
"//myserver.acme.com:5988/root/cimv2:ACME_BooleanKeyClass.KeyProp=True"
"/root/cimv2:ACME_IntegerKeyClass.KeyProp=0x2A"
"ACME_CharKeyClass.KeyProp='\x41'"
```

Instance paths referencing instances of association classes that have key references require special care regarding the escaping of the key values, which in this case are instance paths themselves. As defined in ANNEX A, the `objectHandle` ABNF rule is a string constant whose value conforms to the `objectName` ABNF rule. As defined in 7.11.1, representing a string value as a string in CIM MOF includes the escaping of any double quotes and backslashes present in the string value.

EXAMPLE: The following example shows the string representation of an instance path referencing an instance of an association class with two key references. For better readability, the string is represented in three parts:

```
"/root/cimv2:ACME_SystemDevice."
"System=\"/root/cimv2:ACME_System.Name=\\\"acme\\\""
",Device=\"/root/cimv2:ACME_LogicalDisk.SystemName=\\\"acme\\\",Drive=\\\"C\\\"\""
```

## 8.6 Mapping CIM Naming and Native Naming

A managed environment may identify its managed objects in some way that is not necessarily the way they are identified in their CIM modeled appearance. The identification for managed objects used by the managed environment is called "native naming" in this document.

At the level of interactions between a CIM client and a CIM server, CIM naming is used. This implies that a CIM server needs to be able to map CIM naming to the native naming used by the managed environment. This mapping needs to be performed in both directions: If a CIM operation references an instance with a CIM name, the CIM server needs to map the CIM name into the native name in order to reference the managed object by its native name. Similarly, if a CIM operation requests the enumeration of all instances of a class, the CIM server needs to map the native names by which the managed environment refers to the managed objects, into their CIM names before returning the enumerated instances.

This subclause describes some techniques that can be used by CIM servers to map between CIM names and native names.

### 8.6.1 Native Name Contained in Opaque CIM Key

For CIM classes that have a single opaque key (e.g., InstanceId), it is possible to represent the native name in the opaque key in some (possibly class specific) way. This allows a CIM server to construct the native name from the key value, and vice versa.

### 8.6.2 Native Storage of CIM Name

If the native environment is able to maintain additional properties on its managed objects, the CIM name may be stored on each managed object as an additional property. For larger amounts of instances, this technique requires that there are lookup services available for the CIM server to look up managed objects by CIM name.

### 8.6.3 Translation Table

The CIM server can maintain a translation table between native names and CIM names, which allows to look up the names in both directions. Any entries created in the table are based on a defined mapping between native names and CIM names for the class. The entries in the table are automatically adjusted to the existence of instances as known by the CIM server.

### 8.6.4 No Mapping

Obviously, if the native naming is the same as the CIM naming, then no mapping needs to be performed. This may be the case for environments in which the native representation can be influenced to use CIM naming. An example for that is a relational database, where the relational model is defined such that CIM classes are used as tables, CIM properties as columns, and the index is defined on the columns corresponding to the key properties of the class.

# 9 Mapping Existing Models into CIM

Existing models have their own meta model and model. Three types of mappings can occur between meta schemas: technique, recast, and domain. Each mapping can be applied when MIF syntax is converted to MOF syntax.

## 9.1 Technique Mapping

A technique mapping uses the CIM meta-model constructs to describe the meta constructs of the source modeling technique (for example, MIF, GDMO, and SMI). Essentially, the CIM meta model is a meta meta-model for the source technique (see Figure 9).

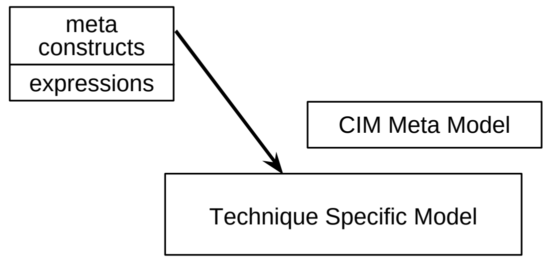



**Figure 9 – Technique Mapping Example**

The DMTF uses the management information format (MIF) as the meta model to describe distributed management information in a common way. Therefore, it is meaningful to describe a technique mapping in which the CIM meta model is used to describe the MIF syntax.

The mapping presented here takes the important types that can appear in a MIF file and then creates classes for them. Thus, component, group, attribute, table, and enum are expressed in the CIM meta model as classes. In addition, associations are defined to document how these classes are combined. Figure 10 illustrates the results.

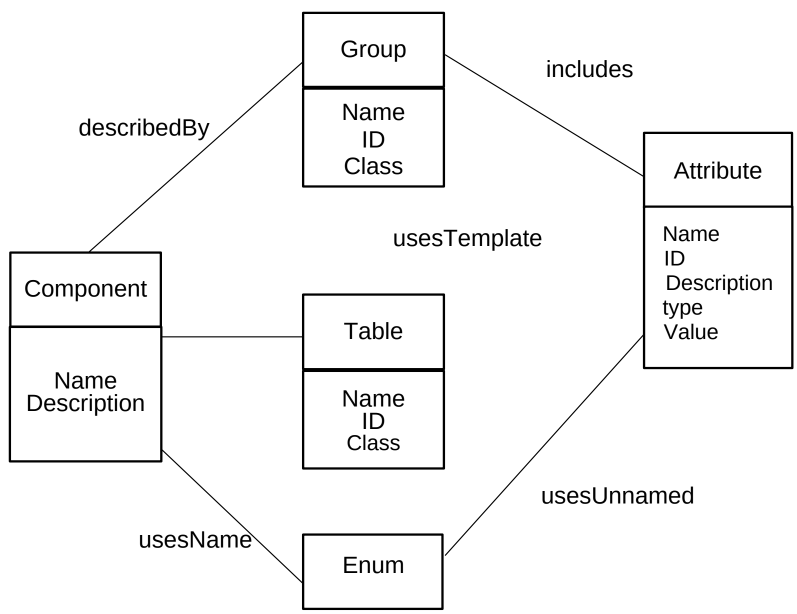



**Figure 10 – MIF Technique Mapping Example**

## 9.2 Recast Mapping

A recast mapping maps the meta constructs of the sources into the targeted meta constructs so that a model expressed in the source can be translated into the target (Figure 11). The major design work is to develop a mapping between the meta model of the sources and the CIM meta model. When this is done, the source expressions are recast.

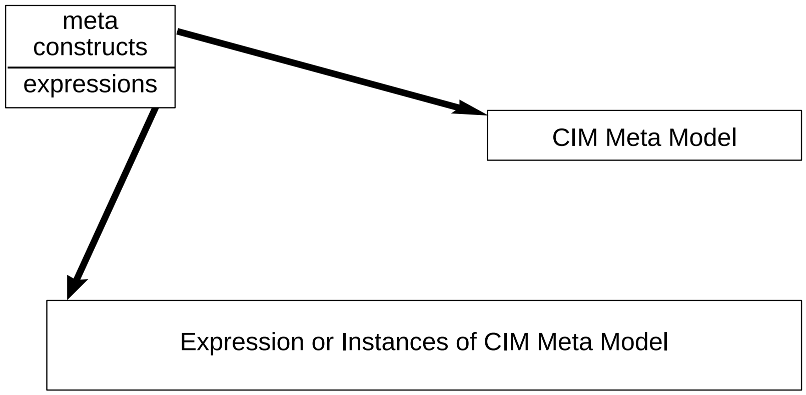



**Figure 11 – Recast Mapping**

Following is an example of a recast mapping for MIF, assuming the following mapping:

DMI attributes -> CIM properties
DMI key attributes -> CIM key properties
DMI groups -> CIM classes
DMI components -> CIM classes

The standard DMI ComponentID group can be recast into a corresponding CIM class:

```
Start Group
Name = "ComponentID"
Class = "DMTF|ComponentID|001"
ID = 1
Description = "This group defines the attributes common to all "
        "components. This group is required."
Start Attribute
    Name = "Manufacturer"
    ID = 1
    Description = "Manufacturer of this system."
    Access = Read-Only
    Storage = Common
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Product"
    ID = 2
    Description = "Product name for this system."
    Access = Read-Only
    Storage = Common
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Version"
    ID = 3
    Description = "Version number of this system."
    Access = Read-Only
    Storage = Specific
```

```
        Type = DisplayString(64)
        Value = ""
End Attribute
Start Attribute
        Name = "Serial Number"
        ID = 4
        Description = "Serial number for this system."
        Access = Read-Only
        Storage = Specific
        Type = DisplayString(64)
        Value = ""
End Attribute
Start Attribute
        Name = "Installation"
        ID = 5
        Description = "Component installation time and date."
        Access = Read-Only
        Storage = Specific
        Type = Date
        Value = ""
End Attribute
Start Attribute
        Name = "Verify"
        ID = 6
        Description = "A code that provides a level of verification that the "
                "component is still installed and working."
        Access = Read-Only
        Storage = Common
        Type = Start ENUM
                0 = "An error occurred; check status code."
                1 = "This component does not exist."
                2 = "Verification is not supported."
                3 = "Reserved."
                4 = "This component exists, but the functionality is untested."
                5 = "This component exists, but the functionality is unknown."
                6 = "This component exists, and is not functioning correctly."
                7 = "This component exists, and is functioning correctly."
        End ENUM
        Value = 1
End Attribute
End Group
```

A corresponding CIM class might be the following. Notice that properties in the example include an ID qualifier to represent the ID of the corresponding DMI attribute. Here, a user-defined qualifier may be necessary:

```
[Name ("ComponentID"), ID (1), Description (
   "This group defines the attributes common to all components. "
   "This group is required.")]
class DMTF|ComponentID|001 {
      [ID (1), Description ("Manufacturer of this system."), maxlen (64)]
   string Manufacturer;
      [ID (2), Description ("Product name for this system."), maxlen (64)]
   string Product;
      [ID (3), Description ("Version number of this system."), maxlen (64)]
   string Version;
```

```
      [ID (4), Description ("Serial number for this system."), maxlen (64)]
   string Serial_Number;
      [ID (5), Description("Component installation time and date.")]
   datetime Installation;
      [ID (6), Description("A code that provides a level of verification "
       "that the component is still installed and working."),
       Value (1)]
   string Verify;
};
```

## 9.3 Domain Mapping

A domain mapping takes a source expressed in a particular technique and maps its content into either the core or common models or extension sub-schemas of the CIM. This mapping does not rely heavily on a meta-to-meta mapping; it is primarily a content-to-content mapping. In one case, the mapping is actually a re-expression of content in a more common way using a more expressive technique.

Following is an example of how DMI can supply CIM properties using information from the DMI disks group ("DMTF|Disks|002"). For a hypothetical CIM disk class, the CIM properties are expressed as shown in Table 11.

**Table 11 – Domain Mapping Example**

| CIM "Disk" Property | Can Be Sourced from DMI Group/Attribute |
|---|---|
| StorageType<br>StorageInterface<br>RemovableDrive<br>RemovableMedia<br>DiskSize | "MIF.DMTF\|Disks\|002.1"<br>"MIF.DMTF\|Disks\|002.3"<br>"MIF.DMTF\|Disks\|002.6"<br>"MIF.DMTF\|Disks\|002.7"<br>"MIF.DMTF\|Disks\|002.16" |

## 9.4 Mapping Scratch Pads

In general, when the contents of models are mapped between different meta schemas, information is lost or missing. To fill this gap, scratch pads are expressed in the CIM meta model using qualifiers, which are actually extensions to the meta model (for example, see 10.2). These scratch pads are critical to the exchange of core, common, and extension model content with the various technologies used to build management applications.

# 10 Repository Perspective

This clause describes a repository and presents a complete picture of the potential to exploit it. A repository stores definitions and structural information, and it includes the capability to extract the definitions in a form that is useful to application developers. Some repositories allow the definitions to be imported into and exported from the repository in multiple forms. The notions of importing and exporting can be refined so that they distinguish between three types of mappings.

Using the mapping definitions in Clause 9, the repository can be organized into the four partitions: meta, technique, recast, and domain (see Figure 12).

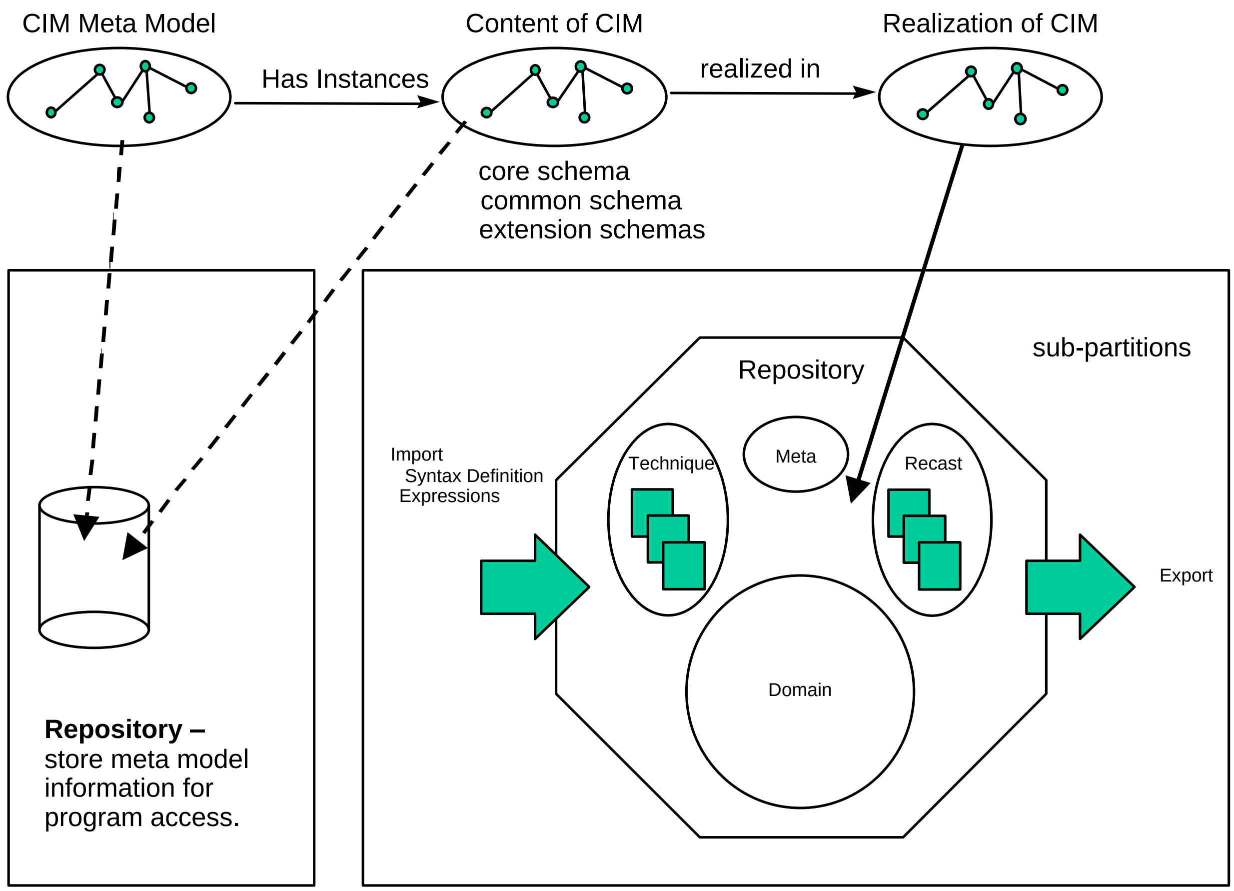



**Figure 12 – Repository Partitions**

The repository partitions have the following characteristics:

- Each partition is discrete:
  - The meta partition refers to the definitions of the CIM meta model.
  - The technique partition refers to definitions that are loaded using technique mappings.
  - The recast partition refers to definitions that are loaded using recast mappings.
  - The domain partition refers to the definitions associated with the core and common models and the extension schemas.
- The technique and recast partitions can be organized into multiple sub-partitions to capture each source uniquely. For example, there is a technique sub-partition for each unique meta language encountered (that is, one for MIF, one for GDMO, one for SMI, and so on). In the re-cast partition, there is a sub-partition for each meta language.
- The act of importing the content of an existing source can result in entries in the recast or domain partition.

## 10.1 DMTF MIF Mapping Strategies

When the meta-model definition and the baseline for the CIM schema are complete, the next step is to map another source of management information (such as standard groups) into the repository. The main goal is to do the work required to import one or more of the standard groups. The possible import scenarios for a DMTF standard group are as follows:

- *To Technique Partition*: Create a technique mapping for the MIF syntax that is the same for all standard groups and needs to be updated only if the MIF syntax changes.
- *To Recast Partition*: Create a recast mapping from a particular standard group into a sub-partition of the recast partition. This mapping allows the entire contents of the selected group to be loaded into a sub-partition of the recast partition. The same algorithm can be used to map additional standard groups into that same sub-partition.
- *To Domain Partition*: Create a domain mapping for the content of a particular standard group that overlaps with the content of the CIM schema.
- *To Domain Partition*: Create a domain mapping for the content of a particular standard group that does not overlap with CIM schema into an extension sub-schema.
- *To Domain Partition*: Propose extensions to the content of the CIM schema and then create a domain mapping.

Any combination of these five scenarios can be initiated by a team that is responsible for mapping an existing source into the CIM repository. Many other details must be addressed as the content of any of the sources changes or when the core or common model changes. When numerous existing sources are imported using all the import scenarios, we must consider the export side. Ignoring the technique partition, the possible export scenarios are as follows:

- *From Recast Partition*: Create a recast mapping for a sub-partition in the recast partition to a standard group (that is, inverse of import 2). The desired method is to use the recast mapping to translate a standard group into a GDMO definition.
- *From Recast Partition*: Create a domain mapping for a recast sub-partition to a known management model that is not the original source for the content that overlaps.
- *From Domain Partition*: Create a recast mapping for the complete contents of the CIM schema to a selected technique (for MIF, this remapping results in a non-standard group).
- *From Domain Partition*: Create a domain mapping for the contents of the CIM schema that overlaps with the content of an existing management model.
- *From Domain Partition*: Create a domain mapping for the entire contents of the CIM schema to an existing management model with the necessary extensions.

## 10.2 Recording Mapping Decisions

To understand the role of the scratch pad in the repository (see Figure 13), it is necessary to look at the import and export scenarios for the different partitions in the repository (technique, recast, and application). These mappings can be organized into two categories: homogeneous and heterogeneous. In the homogeneous category, the imported syntax and expressions are the same as the exported syntax and expressions (for example, software MIF in and software MIF out). In the heterogeneous category, the imported syntax and expressions are different from the exported syntax and expressions (for example, MIF in and GDMO out). For the homogenous category, the information can be recorded by creating qualifiers during an import operation so the content can be exported properly. For the heterogeneous category, the qualifiers must be added after the content is loaded into a partition of the repository. Figure 13 shows the X schema imported into the Y schema and then homogeneously exported into X or heterogeneously exported into Z. Each export arrow works with a different scratch pad.

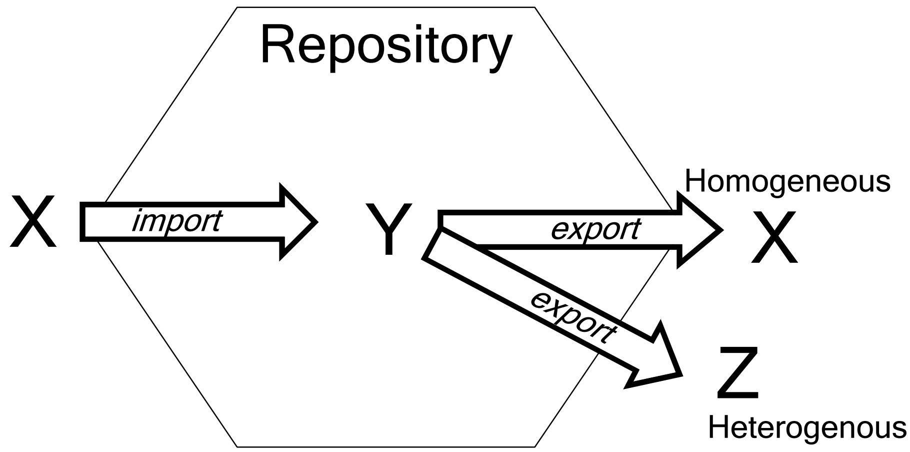



**Figure 13 – Homogeneous and Heterogeneous Export**

The definition of the heterogeneous category is actually based on knowing how a schema is loaded into the repository. To assist in understanding the export process, we can think of this process as using one of multiple scratch pads. One scratch pad is created when the schema is loaded, and the others are added to handle mappings to schema techniques other than the import source (Figure 14).

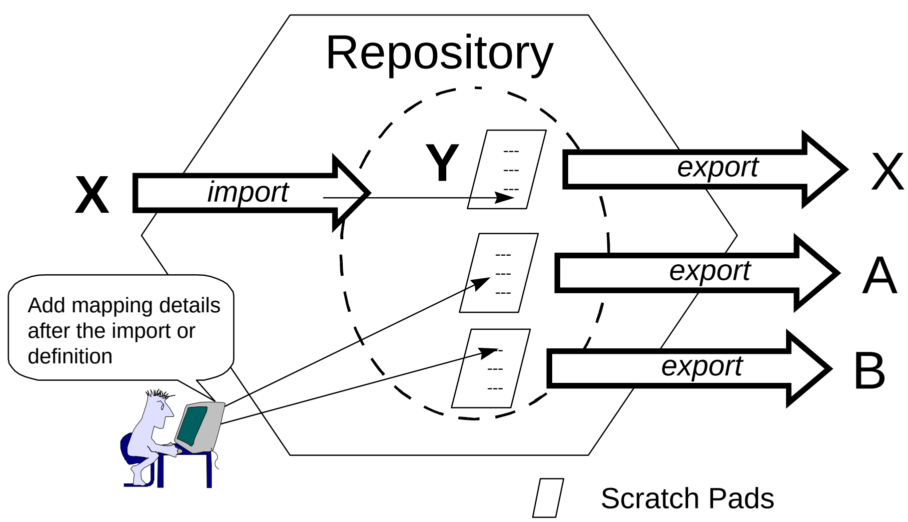



**Figure 14 – Scratch Pads and Mapping**

Figure 14 shows how the scratch pads of qualifiers are used without factoring in the unique aspects of each partition (technique, recast, applications) within the CIM repository. The next step is to consider these partitions.

For the technique partition, there is no need for a scratch pad because the CIM meta model is used to describe the constructs in the source meta schema. Therefore, by definition, there is one homogeneous mapping for each meta schema covered by the technique partition. These mappings create CIM objects for the syntactic constructs of the schema and create associations for the ways they can be combined. (For example, MIF groups include attributes.)

For the recast partition, there are multiple scratch pads for each sub-partition because one is required for each export target and there can be multiple mapping algorithms for each target. Multiple mapping algorithms occur because part of creating a recast mapping involves mapping the constructs of the source into CIM meta-model constructs. Therefore, for the MIF syntax, a mapping must be created for component, group, attribute, and so on, into appropriate CIM meta-model constructs such as object, association, property, and so on. These mappings can be arbitrary. For example, one decision to be made is whether a group or a component maps into an object. Two different recast mapping algorithms are possible: one that maps groups into objects with qualifiers that preserve the component, and one that maps components into objects with qualifiers that preserve the group name for the properties. Therefore, the scratch pads in the recast partition are organized by target technique and employed algorithm.

For the domain partitions, there are two types of mappings:

- A mapping similar to the recast partition in that part of the domain partition is mapped into the syntax of another meta schema. These mappings can use the same qualifier scratch pads and associated algorithms that are developed for the recast partition.
- A mapping that facilitates documenting the content overlap between the domain partition and another model (for example, software groups).

These mappings cannot be determined in a generic way at import time; therefore, it is best to consider them in the context of exporting. The mapping uses filters to determine the overlaps and then performs the necessary conversions. The filtering can use qualifiers to indicate that a particular set of domain partition constructs maps into a combination of constructs in the target/source model. The conversions are documented in the repository using a complex set of qualifiers that capture how to write or insert the overlapped content into the target model. The mapping qualifiers for the domain partition are organized like the recasting partition for the syntax conversions, and there is a scratch pad for each model for documenting overlapping content.

In summary, pick the partition, develop a mapping, and identify the qualifiers necessary to capture potentially lost information when mapping details are developed for a particular source. On the export side, the mapping algorithm verifies whether the content to be exported includes the necessary qualifiers for the logic to work.

# ANNEX A
# (normative)

# MOF Syntax Grammar Description

This annex presents the grammar for MOF syntax. While the grammar is convenient for describing the MOF syntax clearly, the same MOF language can also be described by a different, LL(1)-parsable, grammar, which enables low-footprint implementations of MOF compilers. In addition, the following applies:

1) All keywords are case-insensitive.

2) In the current release, the MOF syntax does not support initializing an array value to empty (an array with no elements). In version 3 of this document, the DMTF plans to extend the MOF syntax to support this functionality as follows:

```
arrayInitialize = "{" [ arrayElementList ] "}"
```

```
arrayElementList = constantValue  *( "," constantValue)
```

To ensure interoperability with implementations of version 2 of this document, the DMTF recommends that, where possible, the value of NULL rather than empty (`{}`) be used to represent the most common use cases. However, if this practice should cause confusion or other issues, implementations may use the syntax of version 3 of this document to initialize an empty array.

The following is the grammar for the MOF syntax, defined in ABNF. Unless otherwise stated, the ABNF in this annex has whitespace allowed.

```
mofSpecification       =   *mofProduction

mofProduction          =   compilerDirective    /
                           classDeclaration     /
                           assocDeclaration     /
                           indicDeclaration     /
                           qualifierDeclaration /
                           instanceDeclaration

compilerDirective      =   PRAGMA pragmaName  "(" pragmaParameter ")"

pragmaName             =   IDENTIFIER

pragmaParameter        =   stringValue

classDeclaration       =   [ qualifierList ]
                           CLASS className  [ superClass ]
                           "{" *classFeature "}" ";"

assocDeclaration       =   "[" ASSOCIATION *( "," qualifier ) "]"
                           CLASS className  [ superClass ]
                           "{" *associationFeature "}" ";"
                           ; Context:
                           ; The remaining qualifier list must not include
                           ; the ASSOCIATION qualifier again. If the
                           ; association has no super association, then at
                           ; least two references must be specified! The
                           ; ASSOCIATION qualifier may be omitted in
                           ; sub-associations.
```

```
indicDeclaration       =   "[" INDICATION *( "," qualifier ) "]"
                           CLASS className  [ superClass ]
                           "{" *classFeature "}" ";"

namespaceName          =   IDENTIFIER *( "/" IDENTIFIER )

className              =   schemaName "_" IDENTIFIER   ; NO whitespace !
                           ; Context:
                           ; Schema name must not include "_" !

alias                  =   AS aliasIdentifer

aliasIdentifer         =   "$" IDENTIFIER   ; NO whitespace !

superClass             =   ":" className

classFeature           =   propertyDeclaration / methodDeclaration

associationFeature     =   classFeature / referenceDeclaration

qualifierList          =   "[" qualifier *( "," qualifier ) "]"

qualifier              =   qualifierName [ qualifierParameter ] [ ":" 1*flavor ]
                           ; DEPRECATED: The ABNF rule [ ":" 1*flavor ] is used
                           ; for the concept of implicitly defined qualifier types
                           ; and is deprecated. See 5.1.2.16 for details.

qualifierParameter     =   "(" constantValue ")" / arrayInitializer

flavor                 =   ENABLEOVERRIDE / DISABLEOVERRIDE / RESTRICTED /
                           TOSUBCLASS / TRANSLATABLE

propertyDeclaration    =   [ qualifierList ] dataType propertyName
                           [ array ] [ defaultValue ] ";"

referenceDeclaration   =   [ qualifierList ] objectRef referenceName
                           [ defaultValue ] ";"

methodDeclaration      =   [ qualifierList ] dataType methodName
                           "(" [ parameterList ] ")" ";"

propertyName           =   IDENTIFIER

referenceName          =   IDENTIFIER

methodName             =   IDENTIFIER

dataType               =   DT_UINT8 / DT_SINT8 / DT_UINT16 / DT_SINT16 /
                           DT_UINT32 / DT_SINT32 / DT_UINT64 / DT_SINT64 /
                           DT_REAL32 / DT_REAL64 / DT_CHAR16 /
                           DT_STR / DT_BOOL / DT_DATETIME

objectRef              =   className REF

parameterList          =   parameter *( "," parameter )

parameter              =   [ qualifierList ] ( dataType / objectRef ) parameterName
                           [ array ]

parameterName          =   IDENTIFIER

array                  =   "[" [positiveDecimalValue] "]"
```

```
positiveDecimalValue  =  positiveDecimalDigit *decimalDigit

defaultValue          =  "=" initializer

initializer           =  ConstantValue / arrayInitializer / referenceInitializer

arrayInitializer      =  "{" constantValue*( "," constantValue)"}"

constantValue         =  integerValue / realValue / charValue / stringValue /
                         datetimeValue / booleanValue / nullValue

integerValue          =  binaryValue / octalValue / decimalValue / hexValue

referenceInitializer  =  objectPath / aliasIdentifier

objectPath            =  stringValue
                         ; the(unescaped)contents of stringValue shall conform
                         ; to the string representation for object paths as
                         ; defined in 8.5.

qualifierDeclaration  =  QUALIFIER qualifierName qualifierType scope
                         [ defaultFlavor ] ";"

qualifierName         =  IDENTIFIER

qualifierType         =  ":" dataType [ array ] [ defaultValue ]

scope                 =  "," SCOPE "(" metaElement *( "," metaElement ) ")"

metaElement           =  CLASS / ASSOCIATION / INDICATION / QUALIFIER
                         PROPERTY / REFERENCE / METHOD / PARAMETER / ANY

defaultFlavor         =  "," FLAVOR "(" flavor *( "," flavor ) ")"

instanceDeclaration   =  [ qualifierList ] INSTANCE OF className [ alias ]
                         "{" 1*valueInitializer "}" ";"

valueInitializer      =  [ qualifierList ]
                         ( propertyName / referenceName ) "=" initializer ";"
```

These ABNF rules do not allow whitespace, unless stated otherwise:

```
schemaName            =  IDENTIFIER
                         ; Context:
                         ; Schema name must not include "_" !

fileName              =  stringValue

binaryValue           =  [ "+" / "-" ] 1*binaryDigit ( "b" / "B" )

binaryDigit           =  "0" / "1"

octalValue            =  [ "+" / "-" ] "0" 1*octalDigit

octalDigit            =  "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"

decimalValue          =  [ "+" / "-" ] ( positiveDecimalDigit *decimalDigit / "0" )

decimalDigit          =  "0" / positiveDecimalDigit

positiveDecimalDigit  =  "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
```

```
hexValue               =   [ "+" / "-" ] ( "0x" / "0X" ) 1*hexDigit

hexDigit               =   decimalDigit / "a" / "A" / "b" / "B" / "c" / "C" /
                           "d" / "D" / "e" / "E" / "f" / "F"

realValue              =   [ "+" / "-" ] *decimalDigit "." 1*decimalDigit
                           [ ( "e" / "E" ) [ "+" / "-" ] 1*decimalDigit ]

charValue              =   "'" char16Char "'" / integerValue
                           ; Single quotes shall be escaped.
                           ; For details, see 7.11.2

stringValue            =   1*( """ *stringChar """ )
                           ; Whitespace and comment is allowed between double
                           ; quoted parts.
                           ; Double quotes shall be escaped.
                           ; For details, see 7.11.1

stringChar             =   UCScharString / stringEscapeSequence

Char16Char             =   UCScharChar16 / stringEscapeSequence

UCScharString              is any UCS character for use in string constants as
                           defined in 7.11.1.

UCScharChar16              is any UCS character for use in char16 constants as
                           defined in 7.11.2.

stringEscapeSequence       is any escape sequence for string and char16 constants, as
                           defined in 7.11.1.

booleanValue           =   TRUE / FALSE

nullValue              =   NULL

IDENTIFIER             =   firstIdentifierChar *( nextIdentifierChar )

firstIdentifierChar    =   UPPERALPHA / LOWERALPHA / UNDERSCORE / UCS0080TOFFEF
                           ; DEPRECATED: The use of the UCS0080TOFFEF ABNF rule
                           ; within the firstIdentifierChar ABNF rule is deprecated
                           ; since version 2.6.0 of this document.

nextIdentifierChar     =   firstIdentifierChar / DIGIT

UPPERALPHA             =   U+0041...U+005A   ; "A" ... "Z"

LOWERALPHA             =   U+0061...U+007A   ; "a" ... "z"

UNDERSCORE             =   U+005F            ; "_"

DIGIT                  =   U+0030...U+0039   ; "0" ... "9"

UCS0080TOFFEF              is any assigned UCS character with code positions in the
                           range U+0080..U+FFEF

datetimeValue          =   1*( """ *stringChar """ )
                           ; Whitespace is allowed between the double quoted parts.
                           ; The combined string value shall conform to the format
                           ; defined by the dt-format ABNF rule.

dt-format              =   dt-timestampValue / dt-intervalValue
```

```
dt-timestampValue      =  14*14(decimalDigit) "." dt-microseconds
                          ("+"/"-") dt-timezone /
                          dt-yyyymmddhhmmss "." 6*6("*") ("+"/"-") dt-timezone
                          ; With further constraints on the field values
                          ; as defined in subclause 5.2.4.

dt-intervalValue       =  14*14(decimalDigit) "." dt-microseconds ":" "000" /
                          dt-dddddddddhhmmss "." 6*6("*") ":" "000"
                          ; With further constraints on the field values
                          ; as defined in subclause 5.2.4.

dt-yyyymmddhhmmss      =  12*12(decimalDigit) 2*2("*") /
                          10*10(decimalDigit) 4*4("*") /
                          8*8(decimalDigit) 6*6("*") /
                          6*6(decimalDigit) 8*8("*") /
                          4*4(decimalDigit) 10*10("*") /
                          14*14("*")

dt-dddddddddhhmmss     =  12*12(decimalDigit) 2*2("*") /
                          10*10(decimalDigit) 4*4("*") /
                          8*8(decimalDigit) 6*6("*") /
                          14*14("*")

dt-microseconds        =  6*6(decimalDigit) /
                          5*5(decimalDigit) 1*1("*") /
                          4*4(decimalDigit) 2*2("*") /
                          3*3(decimalDigit) 3*3("*") /
                          2*2(decimalDigit) 4*4("*") /
                          1*1(decimalDigit) 5*5("*") /
                          6*6("*")

dt-timezone            =  3*3(decimalDigit)
```

4786 The remaining ABNF rules are case-insensitive keywords:

```
ANY                    =  "any"
AS                     =  "as"
ASSOCIATION            =  "association"
CLASS                  =  "class"
DISABLEOVERRIDE        =  "disableOverride"
DT_BOOL                =  "boolean"
DT_CHAR16              =  "char16"
DT_DATETIME            =  "datetime"
DT_REAL32              =  "real32"
DT_REAL64              =  "real64"
DT_SINT16              =  "sint16"
DT_SINT32              =  "sint32"
DT_SINT64              =  "sint64"
DT_SINT8               =  "sint8"
DT_STR                 =  "string"
DT_UINT16              =  "uint16"
DT_UINT32              =  "uint32"
```

```
DT_UINT64          =  "uint64"
DT_UINT8           =  "uint8"
ENABLEOVERRIDE     =  "enableoverride"
FALSE              =  "false"
FLAVOR             =  "flavor"
INDICATION         =  "indication"
INSTANCE           =  "instance"
METHOD             =  "method"
NULL               =  "null"
OF                 =  "of"
PARAMETER          =  "parameter"
PRAGMA             =  "#pragma"
PROPERTY           =  "property"
QUALIFIER          =  "qualifier"
REF                =  "ref"
REFERENCE          =  "reference"
RESTRICTED         =  "restricted"
SCHEMA             =  "schema"
SCOPE              =  "scope"
TOSUBCLASS         =  "tosubclass"
TRANSLATABLE       =  "translatable"
TRUE               =  "true"
```

# ANNEX B
# (informative)

# CIM Meta Schema

This annex defines a CIM model that represents the CIM meta schema defined in 5.1. UML associations are represented as CIM associations.

CIM associations always own their association ends (i.e., the CIM references), while in UML, they are owned either by the association or by the associated class. For sake of simplicity of the description, the UML definition of the CIM meta schema defined in 5.1 had the association ends owned by the associated classes. The CIM model defined in this annex has no other choice but having them owned by the associations. The resulting CIM model is still a correct description of the CIM meta schema.

```
    [Version("2.6.0"), Abstract, Description (
    "Abstract class for CIM elements, providing the ability for "
    "an element to have a name.\n"
    "Some kinds of elements provide the ability to have qualifiers "
    "specified on them, as described in subclasses of "
    "Meta_NamedElement.") ]
class Meta_NamedElement
{
        [Required, Description (
        "The name of the element. The format of the name is "
        "determined by subclasses of Meta_NamedElement.\n"
        "The names of elements shall be compared "
        "case-insensitively.")]
    string Name;
};

// ==================================================================
//    TypedElement
// ==================================================================
    [Version("2.6.0"), Abstract, Description (
    "Abstract class for CIM elements that have a CIM data "
    "type.\n"
    "Not all kinds of CIM data types may be used for all kinds of "
    "typed elements. The details are determined by subclasses of "
    "Meta_TypedElement.") ]
class Meta_TypedElement : Meta_NamedElement
{
};

// ==================================================================
//    Type
// ==================================================================
    [Version("2.6.0"), Abstract, Description (
    "Abstract class for any CIM data types, including arrays of "
    "such."),
```

```
    ClassConstraint {
    "/* If the type is no array type, the value of ArraySize shall "
    "be Null. */\n"
    "inv: self.IsArray = False\n"
    "     implies self.ArraySize.IsNull()"} ]
    "/* A Type instance shall be owned by only one owner. */\n"
    "inv: self.Meta_ElementType[OwnedType].OwningElement->size() +\n"
    "     self.Meta_ValueType[OwnedType].OwningValue->size() = 1"} ]
class Meta_Type
{
        [Required, Description (
        "Indicates whether the type is an array type. For details "
        "on arrays, see 7.8.2.") ]") ]
    boolean IsArray;

        [Description (
        "If the type is an array type, a non-Null value indicates "
        "the size of a fixed-size array, and a Null value indicates "
        "a variable-length array. For details on arrays, see "
        "7.8.2.") ]
    sint64 ArraySize;
};

// ==================================================================
//    PrimitiveType
// ==================================================================
    [Version("2.6.0"), Description (
    "A CIM data type that is one of the intrinsic types defined in "
    "Table 2, excluding references."),
    ClassConstraint {
    "/* This kind of type shall be used only for the following "
    "kinds of typed elements: Method, Parameter, ordinary Property, "
    "and QualifierType. */\n"
    "inv: let e : Meta_NamedElement =\n"
    "       self.Meta_ElementType[OwnedType].OwningElement\n"
    "     in\n"
    "       e.oclIsTypeOf(Meta_Method) or\n"
    "       e.oclIsTypeOf(Meta_Parameter) or\n"
    "       e.oclIsTypeOf(Meta_Property) or\n"
    "       e.oclIsTypeOf(Meta_QualifierType)"} ]
class Meta_PrimitiveType : Meta_Type
{
        [Required, Description (
        "The name of the CIM data type.\n"
        "The type name shall follow the formal syntax defined by "
        "the dataType ABNF rule in ANNEX A.") ]
    string TypeName;
};
```

```
// ==================================================================
//    ReferenceType
// ==================================================================
    [Version("2.6.0"), Description (
    "A CIM data type that is a reference, as defined in Table 2."),
    ClassConstraint {
    "/* This kind of type shall be used only for the following "
    "kinds of typed elements: Parameter and Reference. */\n"
    "inv: let e : Meta_NamedElement = /* the typed element */\n"
    "        self.Meta_ElementType[OwnedType].OwningElement\n"
    "      in\n"
    "        e.oclIsTypeOf(Meta_Parameter) or\n"
    "        e.oclIsTypeOf(Meta_Reference)",
    "/* When used for a Reference, the type shall not be an "
    "array. */\n"
    "inv: self.Meta_ElementType[OwnedType].OwningElement.\n"
    "        oclIsTypeOf(Meta_Reference)\n"
    "      implies\n"
    "        self.IsArray = False"} ]
class Meta_ReferenceType : Meta_Type
{
};
// ==================================================================
//    Schema
// ==================================================================
    [Version("2.6.0"), Description (
    "Models a CIM schema. A CIM schema is a set of CIM classes with "
    "a single defining authority or owning organization."),
    ClassConstraint {
    "/* The elements owned by a schema shall be only of kind "
    "Class. */\n"
    "inv: self.Meta_SchemaElement[OwningSchema].OwnedElement.\n"
    "        oclIsTypeOf(Meta_Class)"} ]
class Meta_Schema : Meta_NamedElement
{
        [Override ("Name"), Description (
        "The name of the schema. The schema name shall follow the "
        "formal syntax defined by the schemaName ABNF rule in "
        "ANNEX A.\n"
        "Schema names shall be compared case insensitively.") ]
    string Name;
};

// ==================================================================
//    Class
// ==================================================================

    [Version("2.6.0"), Description (
    "Models a CIM class. A CIM class is a common type for a set of "
```

```
    "CIM instances that support the same features (i.e. properties "
    "and methods). A CIM class models an aspect of a managed "
    "element.\n"
    "Classes may be arranged in a generalization hierarchy that "
    "represents subtype relationships between classes. The "
    "generalization hierarchy is a rooted, directed graph and "
    "does not support multiple inheritance.\n"
    "A class may have methods, which represent their behavior, "
    "and properties, which represent the data structure of its "
    "instances.\n"
    "A class may participate in associations as the target of a "
    "reference owned by the association.\n"
    "A class may have instances.") ]
class Meta_Class : Meta_NamedElement
{
        [Override ("Name"), Description (
        "The name of the class.\n"
        "The class name shall follow the formal syntax defined by "
        "the className ABNF rule in ANNEX A. The name of "
        "the schema containing the class is part of the class "
        "name.\n"
        "Class names shall be compared case insensitively.\n"
        "The class name shall be unique within the schema owning "
        "the class.") ]
    string Name;
};

// ==================================================================
//    Property
// ==================================================================
    [Version("2.6.0"), Description (
    "Models a CIM property defined in a CIM class. A CIM property "
    "is the declaration of a structural feature of a CIM class, "
    "i.e. the data structure of its instances.\n"
    "Properties are inherited to subclasses such that instances of "
    "the subclasses have the inherited properties in addition to "
    "the properties defined in the subclass. The combined set of "
    "properties defined in a class and properties inherited from "
    "superclasses is called the properties exposed by the class.\n"
    "A class defining a property may indicate that the property "
    "overrides an inherited property. In this case, the class "
    "exposes only the overriding property. The characteristics of "
    "the overriding property are formed by using the "
    "characteristics of the overridden property as a basis, "
    "changing them as defined in the overriding property, within "
    "certain limits as defined in additional constraints.\n"
    "The class owning an overridden property shall be a (direct "
    "or indirect) superclass of the class owning the overriding "
    "property.\n"
```

```
    "For references, the class referenced by the overriding "
    "reference shall be the same as, or a subclass of, the class "
    "referenced by the overridden reference."),
    ClassConstraint {
    "/* An overriding property shall have the same name as the "
    "property it overrides. */\n"
    "inv: self.Meta_PropertyOverride[OverridingProperty]->\n"
    "       size() = 1\n"
    "     implies\n"
    "     self.Meta_PropertyOverride[OverridingProperty].\n"
    "       OverriddenProperty.Name.toUpper() =\n"
    "     self.Name.toUpper()",
    "/* For ordinary properties, the data type of the overriding "
    "property shall be the same as the data type of the overridden "
    "property. */\n"
    "inv: self.oclIsTypeOf(Meta_Property) and\n"
    "       Meta_PropertyOverride[OverridingProperty]->\n"
    "       size() = 1\n"
    "     implies\n"
    "       let pt : Meta_Type = /* type of property */\n"
    "         self.Meta_ElementType[Element].Type\n"
    "       in\n"
    "       let opt : Meta_Type = /* type of overridden prop. */\n"
    "         self.Meta_PropertyOverride[OverridingProperty].\n"
    "         OverriddenProperty.Meta_ElementType[Element].Type\n"
    "       in\n"
    "       opt.TypeName.toUpper() = pt.TypeName.toUpper() and\n"
    "       opt.IsArray   = pt.IsArray   and\n"
    "       opt.ArraySize = pt.ArraySize"} ]
class Meta_Property : Meta_TypedElement
{
        [Override ("Name"), Description (
        "The name of the property. The property name shall follow "
        "the formal syntax defined by the propertyName ABNF rule "
        "in ANNEX A.\n"
        "Property names shall be compared case insensitively.\n"
        "Property names shall be unique within its owning (i.e. "
        "defining) class.\n"
        "NOTE: The set of properties exposed by a class may have "
        "duplicate names if a class defines a property with the "
        "same name as a property it inherits without overriding "
        "it.") ]
    string Name;

        [Description (
        "The default value of the property, in its string "
        "representation.") ]
    string DefaultValue [];
};
```

```

// ==================================================================
//    Method
// ==================================================================

    [Version("2.6.0"), Description (
    "Models a CIM method. A CIM method is the declaration of a "
    "behavioral feature of a CIM class, representing the ability "
    "for invoking an associated behavior.\n"
    "The CIM data type of the method defines the declared return "
    "type of the method.\n"
    "Methods are inherited to subclasses such that subclasses have "
    "the inherited methods in addition to the methods defined in "
    "the subclass. The combined set of methods defined in a class "
    "and methods inherited from superclasses is called the methods "
    "exposed by the class.\n"
    "A class defining a method may indicate that the method "
    "overrides an inherited method. In this case, the class exposes "
    "only the overriding method. The characteristics of the "
    "overriding method are formed by using the characteristics of "
    "the overridden method as a basis, changing them as defined in "
    "the overriding method, within certain limits as defined in "
    "additional constraints.\n"
    "The class owning an overridden method shall be a superclass "
    "of the class owning the overriding method."),
    ClassConstraint {
    "/* An overriding method shall have the same name as the "
    "method it overrides. */\n"
    "inv: self.Meta_MethodOverride[OverridingMethod]->\n"
    "       size() = 1\n"
    "     implies\n"
    "       self.Meta_MethodOverride[OverridingMethod].\n"
    "         OverriddenMethod.Name.toUpper() =\n"
    "       self.Name.toUpper()",
    "/* The return type of a method shall not be an array. */\n"
    "inv: self.Meta_ElementType[Element].Type.IsArray = False",
    "/* An overriding method shall have the same signature "
    "(i.e. parameters and return type) as the method it "
    "overrides. */\n"
    "inv: Meta_MethodOverride[OverridingMethod]->size() = 1\n"
    "     implies\n"
    "       let om : Meta_Method = /* overridden method */\n"
    "         self.Meta_MethodOverride[OverridingMethod].\n"
    "           OverriddenMethod\n"
    "       in\n"
    "       om.Meta_ElementType[Element].Type.TypeName.toUpper() =\n"
    "         self.Meta_ElementType[Element].Type.TypeName.toUpper()\n"
    "       and\n"
    "       Set {1 .. om.Meta_MethodParameter[OwningMethod].\n"
```

```
    "            OwnedParameter->size()}\n"
    "       ->forAll( i |\n"
    "         let omp : Meta_Parameter = /* parm in overridden method */\n"
    "           om.Meta_MethodParameter[OwningMethod].OwnedParameter->\n"
    "             asOrderedSet()->at(i)\n"
    "         in\n"
    "         let selfp : Meta_Parameter = /* parm in overriding method */\n"
    "           self.Meta_MethodParameter[OwningMethod].OwnedParameter->\n"
    "             asOrderedSet()->at(i)\n"
    "         in\n"
    "         omp.Name.toUpper() = selfp.Name.toUpper() and\n"
    "         omp.Meta_ElementType[Element].Type.TypeName.toUpper() =\n"
    "           selfp.Meta_ElementType[Element].Type.TypeName.toUpper()\n"
    "       )"} ]
class Meta_Method : Meta_TypedElement
{
        [Override ("Name"), Description (
        "The name of the method. The method name shall follow "
        "the formal syntax defined by the methodName ABNF rule in "
        "ANNEX A.\n"
        "Method names shall be compared case insensitively.\n"
        "Method names shall be unique within its owning (i.e. "
        "defining) class.\n"
        "NOTE: The set of methods exposed by a class may have "
        "duplicate names if a class defines a method with the same "
        "name as a method it inherits without overriding it.") ]
    string Name;
};

// ==================================================================
//    Parameter
// ==================================================================
    [Version("2.6.0"), Description (
    "Models a CIM parameter. A CIM parameter is the declaration of "
    "a parameter of a CIM method. The return value of a "
    "method is not modeled as a parameter.") ]
class Meta_Parameter : Meta_TypedElement
{
        [Override ("Name"), Description (
        "The name of the parameter. The parameter name shall follow "
        "the formal syntax defined by the parameterName ABNF rule "
        "in ANNEX A.\n"
        "Parameter names shall be compared case insensitively.") ]
    string Name;
};

// ==================================================================
//    Trigger
// ==================================================================
```

```
    [Version("2.6.0"), Description (
    "Models a CIM trigger. A CIM trigger is the specification of a "
    "rule on a CIM element that defines when the trigger is to be "
    "fired.\n"
    "Triggers may be fired on the following occasions:\n"
    "* On creation, deletion, modification, or access of CIM "
    "instances of ordinary classes and associations. The trigger is "
    "specified on the class in this case and applies to all "
    "instances.\n"
    "* On modification, or access of a CIM property. The trigger is "
    "specified on the property in this case and and applies to all "
    "instances.\n"
    "* Before and after the invocation of a CIM method. The trigger "
    "is specified on the method in this case and and applies to all "
    "invocations of the method.\n"
    "* When a CIM indication is raised. The trigger is specified on "
    "the indication in this case and and applies to all occurences "
    "for when this indication is raised.\n"
    "The rules for when a trigger is to be fired are specified with "
    "the TriggerType qualifier.\n"
    "The firing of a trigger shall cause the indications to be "
    "raised that are associated to the trigger via "
    "Meta_TriggeredIndication."),
    ClassConstraint {
    "/* Triggers shall be specified only on ordinary classes, "
    "associations, properties (including references), methods and "
    "indications. */\n"
    "inv: let e : Meta_NamedElement = /* the element on which\n"
    "                             the trigger is specified */\n"
    "        self.Meta_TriggeringElement[Trigger].Element\n"
    "      in\n"
    "        e.oclIsTypeOf(Meta_Class) or\n"
    "        e.oclIsTypeOf(Meta_Association) or\n"
    "        e.oclIsTypeOf(Meta_Property) or\n"
    "        e.oclIsTypeOf(Meta_Reference) or\n"
    "        e.oclIsTypeOf(Meta_Method) or\n"
    "        e.oclIsTypeOf(Meta_Indication)"} ]
class Meta_Trigger : Meta_NamedElement
{
        [Override ("Name"), Description (
        "The name of the trigger.\n"
        "Trigger names shall be compared case insensitively.\n"
        "Trigger names shall be unique "
        "within the property, class or method to which the trigger "
        "applies.") ]
    string Name;
};
```

```
// ==================================================================
//    Indication
// ==================================================================

    [Version("2.6.0"), Description (
    "Models a CIM indication. An instance of a CIM indication "
    "represents an event that has occurred. If an instance of an "
    "indication is created, the indication is said to be raised. "
    "The event causing an indication to be raised may be that a "
    "trigger has fired, but other arbitrary events may cause an "
    "indication to be raised as well."),
    ClassConstraint {
    "/* An indication shall not own any methods. */\n"
    "inv: self.MethodDomain[OwningClass].OwnedMethod->size() = 0"} ]
class Meta_Indication : Meta_Class
{
};

// ==================================================================
//    Association
// ==================================================================

    [Version("2.6.0"), Description (
    "Models a CIM association. A CIM association is a special kind "
    "of CIM class that represents a relationship between two or more "
    "CIM classes. A CIM association owns its association ends (i.e. "
    "references). This allows for adding associations to a schema "
    "without affecting the associated classes."),
    ClassConstraint {
    "/* The superclass of an association shall be an association. */\n"
    "inv: self.Meta_Generalization[SubClass].SuperClass->\n"
    "       oclIsTypeOf(Meta_Association)",
    "/* An association shall own two or more references. */\n"
    "inv: self.Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
    "       select( p | p.oclIsTypeOf(Meta_Reference))->size() >= 2",
    "/* The number of references exposed by an association (i.e. "
    "its arity) shall not change in its subclasses. */\n"
    "inv: self.Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
    "       select( p | p.oclIsTypeOf(Meta_Reference))->size() =\n"
    "     self.Meta_Generalization[SubClass].SuperClass->\n"
    "       Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
    "       select( p | p.oclIsTypeOf(Meta_Reference))->size()"} ]
class Meta_Association : Meta_Class
{
};

// ==================================================================
//    Reference
// ==================================================================
```

```

    [Version("2.6.0"), Description (
    "Models a CIM reference. A CIM reference is a special kind of "
    "CIM property that represents an association end, as well as a "
    "role the referenced class plays in the context of the "
    "association owning the reference."),
    ClassConstraint {
    "/* A reference shall be owned by an association (i.e. not "
    "by an ordinary class or by an indication). As a result "
    "of this, reference names do not need to be unique within any "
    "of the associated classes. */\n"
    "inv: self.Meta_PropertyDomain[OwnedProperty].OwningClass.\n"
    "       oclIsTypeOf(Meta_Association)"} ]
class Meta_Reference : Meta_Property
{
        [Override ("Name"), Description (
        "The name of the reference. The reference name shall follow "
        "the formal syntax defined by the referenceName ABNF rule "
        "in ANNEX A.\n"
        "Reference names shall be compared case insensitively.\n"
        "Reference names shall be unique within its owning (i.e. "
        "defining) association.") ]
    string Name;
};

// ==================================================================
//    QualifierType
// ==================================================================
    [Version("2.6.0"), Description (
    "Models the declaration of a CIM qualifier (i.e. a qualifier "
    "type). A CIM qualifier is meta data that provides additional "
    "information about the element on which the qualifier is "
    "specified.\n"
    "The qualifier type is either explicitly defined in the CIM "
    "namespace, or implicitly defined on an element as a result of "
    "a qualifier that is specified on an element for which no "
    "explicit qualifier type is defined.\n"
    "Implicitly defined qualifier types shall agree in data type, "
    "scope, flavor and default value with any explicitly defined "
    "qualifier types of the same name. \n
    "DEPRECATED: The concept of implicitly defined qualifier "
    "types is deprecated.") ]
class Meta_QualifierType : Meta_TypedElement
{
        [Override ("Name"), Description (
        "The name of the qualifier. The qualifier name shall follow "
        "the formal syntax defined by the qualifierName ABNF rule "
        "in ANNEX A.\n"
        "The names of explicitly defined qualifier types shall be "
```

```
        "unique within the CIM namespace. Unlike classes, "
        "qualifier types are not part of a schema, so name "
        "uniqueness cannot be defined at the definition level "
        "relative to a schema, and is instead only defined at "
        "the object level relative to a namespace.\n"
        "The names of implicitly defined qualifier types shall be "
        "unique within the scope of the CIM element on which the "
        "qualifiers are specified.") ]
    string Name;

        [Description (
        "The scopes of the qualifier. The qualifier scopes determine "
        "to which kinds of elements a qualifier may be specified on. "
        "Each qualifier scope shall be one of the following keywords:\n"
        "  \"any\" - the qualifier may be specified on any qualifiable element.\n"
        "  \"class\" - the qualifier may be specified on any ordinary class.\n"
        "  \"association\" - the qualifier may be specified on any association.\n"
        "  \"indication\" - the qualifier may be specified on any indication.\n"
        "  \"property\" - the qualifier may be specified on any ordinary property.\n"
        "  \"reference\" - the qualifier may be specified on any reference.\n"
        "  \"method\" - the qualifier may be specified on any method.\n"
        "  \"parameter\" - the qualifier may be specified on any parameter.\n"
        "Qualifiers cannot be specified on qualifiers.") ]
    string Scope [];
};

// ==================================================================
//    Qualifier
// ==================================================================

    [Version("2.6.0"), Description (
    "Models the specification (i.e. usage) of a CIM qualifier on an "
    "element. A CIM qualifier is meta data that provides additional "
    "information about the element on which the qualifier is "
    "specified. The specification of a qualifier on an element "
    "defines a value for the qualifier on that element.\n"
    "If no explicitly defined qualifier type exists with this name "
    "in the CIM namespace, the specification of a qualifier causes an "
    "implicitly defined qualifier type (i.e. a Meta_QualifierType "
    "element) to be created on the qualified element. \n
    "DEPRECATED: The concept of implicitly defined qualifier "
    "types is deprecated.") ]
class Meta_Qualifier : Meta_NamedElement
{
        [Override ("Name"), Description (
        "The name of the qualifier. The qualifier name shall follow "
        "the formal syntax defined by the qualifierName ABNF rule "
        "in ANNEX A. \n
        "The names of explicitly defined qualifier types shall be "
```

```
        "unique within the CIM namespace. Unlike classes, "
        "qualifier types are not part of a schema, so name "
        "uniqueness cannot be defined at the definition level "
        "relative to a schema, and is instead only defined at "
        "the object level relative to a namespace.\n"
        "The names of implicitly defined qualifier types shall be "
        "unique within the scope of the CIM element on which the "
        "qualifiers are specified." \n
        "DEPRECATED: The concept of implicitly defined qualifier "
        "types is deprecated.") ]
    string Name;

        [Description (
        "The scopes of the qualifier. The qualifier scopes determine "
        "to which kinds of elements a qualifier may be specified on. "
        "Each qualifier scope shall be one of the following keywords:\n"
        "  \"any\" - the qualifier may be specified on any qualifiable element.\n"
        "  \"class\" - the qualifier may be specified on any ordinary class.\n"
        "  \"association\" - the qualifier may be specified on any association.\n"
        "  \"indication\" - the qualifier may be specified on any indication.\n"
        "  \"property\" - the qualifier may be specified on any ordinary property.\n"
        "  \"reference\" - the qualifier may be specified on any reference.\n"
        "  \"method\" - the qualifier may be specified on any method.\n"
        "  \"parameter\" - the qualifier may be specified on any parameter.\n"
        "Qualifiers cannot be specified on qualifiers.") ]
    string Scope [];
};

// ==================================================================
//    Flavor
// ==================================================================
    [Version("2.6.0"), Description (
    "The specification of certain characteristics of the qualifier "
    "such as its value propagation from the ancestry of the "
    "qualified element, and translatability of the qualifier "
    "value.") ]
class Meta_Flavor
{
        [Description (
        "Indicates whether the qualifier value is to be propagated "
        "from the ancestry of an element in case the qualifier is "
        "not specified on the element.") ]
    boolean InheritancePropagation;

        [Description (
        "Indicates whether qualifier values propagated to an "
        "element may be overridden by the specification of that "
        "qualifier on the element.") ]
    boolean OverridePermission;
```

```
        [Description (
        "Indicates whether qualifier value is translatable.") ]
    boolean Translatable;
};

// ==================================================================
//    Instance
// ==================================================================
    [Version("2.6.0"), Description (
    "Models a CIM instance. A CIM instance is an instance of a CIM "
    "class that specifies values for a subset (including all) of the "
    "properties exposed by its defining class.\n"
    "A CIM instance in a CIM server shall have exactly the properties "
    "exposed by its defining class.\n"
    "A CIM instance cannot redefine the properties "
    "or methods exposed by its defining class and cannot have "
    "qualifiers specified.\n"
    "A particular property shall be specified at most once in a "
    "given instance.") ]
class Meta_Instance
{
};

// ==================================================================
//    InstanceProperty
// ==================================================================
    [Version("2.6.0"), Description (
    "The definition of a property value within a CIM instance.") ]
class Meta_InstanceProperty
{
};

// ==================================================================
//    Value
// ==================================================================
    [Version("2.6.0"), Description (
    "A typed value, used in several contexts."),
    ClassConstraint {
    "/* If the Null indicator is set, no values shall be specified. "
    "*/\n"
    "inv: self.IsNull = True\n"
    "     implies self.Value->size() = 0",
    "/* If values are specified, the Null indicator shall not be "
    "set. */\n"
    "inv: self.Value->size() > 0\n"
    "     implies self.IsNull = False",
    "/* A Value instance shall be owned by only one owner. */\n"
    "inv: self.OwningProperty->size() +\n"
```

```
    "     self.OwningInstanceProperty->size() +\n"
    "     self.OwningQualifierType->size() +\n"
    "     self.OwningQualifier->size() = 1"} ]
class Meta_Value
{
        [Description (
        "The scalar value or the array of values. "
        "Each value is represented as a string.") ]
    string Value [];

        [Description (
        "The Null indicator of the value. "
        "If True, the value is Null. "
        "If False, the value is indicated through the Value "
        attribute.") ]
    boolean IsNull;
};

// ==================================================================
//    SpecifiedQualifier
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_SpecifiedQualifier
{
        [Aggregate, Min (1), Max (1), Description (
        "The element on which the qualifier is specified.") ]
    Meta_NamedElement REF OwningElement;

        [Min (0), Max (Null), Description (
        "The qualifier specified on the element.") ]
    Meta_Qualifier REF OwnedQualifier;
};

// ==================================================================
//    ElementType
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_ElementType
{
        [Aggregate, Min (0), Max (1), Description (
        "The element that has a CIM data type.") ]
    Meta_TypedElement REF OwningElement;

        [Min (1), Max (1), Description (
        "The CIM data type of the element.") ]
    Meta_Type REF OwnedType;
};

// ==================================================================
```

```
//    PropertyDomain
// ==================================================================

    [Association, Composition, Version("2.6.0")]
class Meta_PropertyDomain
{
        [Aggregate, Min (1), Max (1), Description (
        "The class owning (i.e. defining) the property.") ]
     Meta_Class REF OwningClass;

        [Min (0), Max (Null), Description (
        "The property owned by the class.") ]
    Meta_Property REF OwnedProperty;
};

// ==================================================================
//    MethodDomain
// ==================================================================

    [Association, Composition, Version("2.6.0")]
class Meta_MethodDomain
{
        [Aggregate, Min (1), Max (1), Description (
        "The class owning (i.e. defining) the method.") ]
     Meta_Class REF OwningClass;

        [Min (0), Max (Null), Description (
        "The method owned by the class.") ]
    Meta_Method REF OwnedMethod;
};

// ==================================================================
//    ReferenceRange
// ==================================================================

    [Association, Version("2.6.0")]
class Meta_ReferenceRange
{
        [Min (0), Max (Null), Description (
        "The reference type referencing the class.") ]
    Meta_ReferenceType REF ReferencingType;

        [Min (1), Max (1), Description (
        "The class referenced by the reference type.") ]
     Meta_Class REF ReferencedClass;
};

// ==================================================================
//    QualifierTypeFlavor
```

```
// ==================================================================

    [Association, Composition, Version("2.6.0")]
class Meta_QualifierTypeFlavor
{
        [Aggregate, Min (1), Max (1), Description (
        "The qualifier type defining the flavor.") ]
    Meta_QualifierType REF QualifierType;

        [Min (1), Max (1), Description (
        "The flavor of the qualifier type.") ]
     Meta_Flavor REF Flavor;
};

// ==================================================================
//    Generalization
// ==================================================================

    [Association, Version("2.6.0")]
class Meta_Generalization
{
        [Min (0), Max (Null), Description (
        "The subclass of the class.") ]
    Meta_Class REF SubClass;

        [Min (0), Max (1), Description (
        "The superclass of the class.") ]
     Meta_Class REF SuperClass;
};

// ==================================================================
//    PropertyOverride
// ==================================================================

    [Association, Version("2.6.0")]
class Meta_PropertyOverride
{
        [Min (0), Max (Null), Description (
        "The property overriding this property.") ]
    Meta_Property REF OverridingProperty;

        [Min (0), Max (1), Description (
        "The property overridden by this property.") ]
     Meta_Property REF OverriddenProperty;
};

// ==================================================================
//    MethodOverride
// ==================================================================
```

```
    [Association, Version("2.6.0")]
class Meta_MethodOverride
{
        [Min (0), Max (Null), Description (
        "The method overriding this method.") ]
    Meta_Method REF OverridingMethod;

        [Min (0), Max (1), Description (
        "The method overridden by this method.") ]
     Meta_Method REF OverriddenMethod;
};

// ==================================================================
//    SchemaElement
// ==================================================================

    [Association, Composition, Version("2.6.0")]
class Meta_SchemaElement
{
        [Aggregate, Min (1), Max (1), Description (
        "The schema owning the element.") ]
    Meta_Schema REF OwningSchema;

        [Min (0), Max (Null), Description (
        "The elements owned by the schema.") ]
     Meta_NamedElement REF OwnedElement;
};

// ==================================================================
//    MethodParameter
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_MethodParameter
{
        [Aggregate, Min (1), Max (1), Description (
        "The method owning (i.e. defining) the parameter.") ]
    Meta_Method REF OwningMethod;

        [Min (0), Max (Null), Description (
        "The parameter of the method. The return value "
        "is not represented as a parameter.") ]
    Meta_Parameter REF OwnedParameter;
};

// ==================================================================
//    SpecifiedProperty
// ==================================================================
    [Association, Composition, Version("2.6.0")]
```

```
class Meta_SpecifiedProperty
{
        [Aggregate, Min (1), Max (1), Description (
        "The instance for which a property value is defined.") ]
    Meta_Instance REF OwningInstance;

        [Min (0), Max (Null), Description (
        "The property value specified by the instance.") ]
    Meta_PropertyValue REF OwnedPropertyValue;
};

// ==================================================================
//    DefiningClass
// ==================================================================
    [Association, Version("2.6.0")]
class Meta_DefiningClass
{
        [Min (0), Max (Null), Description (
        "The instances for which the class is their defining class.") ]
    Meta_Instance REF Instance;

        [Min (1), Max (1), Description (
        "The defining class of the instance.") ]
    Meta_Class REF DefiningClass;
};

// ==================================================================
//    DefiningQualifier
// ==================================================================
    [Association, Version("2.6.0")]
class Meta_DefiningQualifier
{
        [Min (0), Max (Null), Description (
        "The specification (i.e. usage) of the qualifier.") ]
    Meta_Qualifier REF Qualifier;

        [Min (1), Max (1), Description (
        "The qualifier type defining the characteristics of the "
        "qualifier.") ]
    Meta_QualifierType REF QualifierType;
};

// ==================================================================
//    DefiningProperty
// ==================================================================
    [Association, Version("2.6.0")]
class Meta_DefiningProperty
{
        [Min (1), Max (1), Description (
```

```
        "A value of this property in an instance.") ]
    Meta_PropertyValue REF InstanceProperty;

        [Min (0), Max (Null), Description (
        "The declaration of the property for which a value is "
        "defined.") ]
    Meta_Property REF DefiningProperty;
};

// ==================================================================
//    ElementQualifierType
// ==================================================================
    [Association, Version("2.6.0"), Description (
        "DEPRECATED: The concept of implicitly defined qualifier "
        "types is deprecated.") ]
class Meta_ElementQualifierType
{
        [Min (0), Max (1), Description (
        "For implicitly defined qualifier types, the element on "
        "which the qualifier type is defined.\n"
        "Qualifier types defined explicitly are not "
        "associated to elements, they are global in the CIM "
        "namespace.") ]
    Meta_NamedElement REF Element;

        [Min (0), Max (Null), Description (
        "The qualifier types implicitly defined on the element.\n"
        "Qualifier types defined explicitly are not "
        "associated to elements, they are global in the CIM "
        "namespace.") ]
    Meta_QualifierType REF QualifierType;
};

// ==================================================================
//    TriggeringElement
// ==================================================================
    [Association, Version("2.6.0")]
class Meta_TriggeringElement
{
        [Min (0), Max (Null), Description (
        "The triggers specified on the element.") ]
    Meta_Trigger REF Trigger;

        [Min (1), Max (Null), Description (
        "The CIM element on which the trigger is specified.") ]
    Meta_NamedElement REF Element;
};

// ==================================================================
```

```
//    TriggeredIndication
// ==================================================================
    [Association, Version("2.6.0")]
class Meta_TriggeredIndication
{
        [Min (0), Max (Null), Description (
        "The triggers specified on the element.") ]
    Meta_Trigger REF Trigger;

        [Min (0), Max (Null), Description (
        "The CIM element on which the trigger is specified.") ]
    Meta_Indication REF Indication;
};
// ==================================================================
//    ValueType
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_ValueType
{
        [Aggregate, Min (0), Max (1), Description (
        "The value that has a CIM data type.") ]
    Meta_Value REF OwningValue;

        [Min (1), Max (1), Description (
        "The type of this value.") ]
    Meta_Type REF OwnedType;
};

// ==================================================================
//    PropertyDefaultValue
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_PropertyDefaultValue
{
        [Aggregate, Min (0), Max (1), Description (
        "A property declaration that defines this value as its "
        "default value.") ]
    Meta_Property REF OwningProperty;

        [Min (0), Max (1), Description (
        "The default value of the property declaration. A Value "
        "instance shall be associated if and only if a default "
        "value is defined on the property declaration.") ]
    Meta_Value REF OwnedDefaultValue;
};

// ==================================================================
//    QualifierTypeDefaultValue
// ==================================================================
```

```
    [Association, Composition, Version("2.6.0")]
class Meta_QualifierTypeDefaultValue
{
        [Aggregate, Min (0), Max (1), Description (
        "A qualifier type declaration that defines this value as "
        "its default value.") ]
    Meta_QualifierType REF OwningQualifierType;

        [Min (0), Max (1), Description (
        "The default value of the qualifier declaration. A Value "
        "instance shall be associated if and only if a default "
        "value is defined on the qualifier declaration.") ]
    Meta_Value REF OwnedDefaultValue;
};

// ==================================================================
//    PropertyValue
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_PropertyValue
{
        [Aggregate, Min (0), Max (1), Description (
        "A property defined in an instance that has this value.") ]
    Meta_InstanceProperty REF OwningInstanceProperty;

        [Min (1), Max (1), Description (
        "The value of the property.") ]
    Meta_Value REF OwnedValue;

// ==================================================================
//    QualifierValue
// ==================================================================
    [Association, Composition, Version("2.6.0")]
class Meta_QualifierValue
{
        [Aggregate, Min (0), Max (1), Description (
        "A qualifier defined on a schema element that has this "
        "value.") ]
    Meta_Qualifier REF OwningQualifier;

        [Min (1), Max (1), Description (
        "The value of the qualifier.") ]
    Meta_Value REF OwnedValue;
};
```

# ANNEX C
# (normative)

# Units

## C.1 Programmatic Units

This annex defines the concept and syntax of a programmatic unit, which is an expression of a unit of measure for programmatic access. It makes it easy to recognize the base units of which the actual unit is made, as well as any numerical multipliers. Programmatic units are used as a value for the PUnit qualifier and also as a value for any (string typed) CIM elements that represent units. The boolean IsPUnit qualifier is used to declare that a string typed element follows the syntax for programmatic units.

Programmatic units must be processed case-sensitively and white-space-sensitively.

As defined in the Augmented BNF (ABNF) syntax, the programmatic unit consists of a base unit that is optionally followed by other base units that are each either multiplied or divided into the first base unit. Furthermore, two optional multipliers can be applied. The first is simply a scalar, and the second is an exponential number consisting of a base and an exponent. The optional multipliers enable the specification of common derived units of measure in terms of the allowed base units. The base units defined in this subclause include a superset of the SI base units. When a unit is the empty string, the value has no unit; that is, it is dimensionless. The multipliers must be understood as part of the definition of the derived unit; that is, scale prefixes of units are replaced with their numerical value. For example, "kilometer" is represented as "meter * 1000", replacing the "kilo" scale prefix with the numerical factor 1000.

A string representing a programmatic unit must follow the format defined by the `programmatic-unit` ABNF rule in the syntax defined in this annex. This format supports any type of unit, including SI units, United States units, and any other standard or non-standard units.

The ABNF syntax is defined as follows. This ABNF explicitly states any whitespace characters that may be used, and whitespace characters in addition to those are not allowed.

```
programmatic-unit = ( "" / base-unit  *( [WS] multiplied-base-unit )
                  *( [WS] divided-base-unit )  [ [WS] modifier1]  [ [WS] modifier2 ] )

multiplied-base-unit = "*" [WS] base-unit

divided-base-unit = "/" [WS] base-unit

modifier1 = operator [WS] number

modifier2 = operator [WS] base [WS] "^" [WS] exponent

operator = "*" / "/"

number = ["+" / "-"] positive-number

base = positive-whole-number

exponent = ["+" / "-"] positive-whole-number
```

```
positive-whole-number = NON-ZERO-DIGIT *( DIGIT )

positive-number = positive-whole-number
                / ( ( positive-whole-number / ZERO ) "." *( DIGIT ) )

base-unit = simple-name / decibel-base-unit

simple-name = FIRST-UNIT-CHAR *( [S] UNIT-CHAR )

decibel-base-unit = "decibel" [ [S] "(" [S] simple-name [S] ")" ]

FIRST-UNIT-CHAR = UPPERALPHA / LOWERALPHA / UNDERSCORE / UCS0080TOFFEF
                 ; DEPRECATED: The use of the UCS0080TOFFEF ABNF rule within
                 ; the FIRST-UNIT-CHAR ABNF rule is deprecated since
                 ; version 2.6.0 of this document.

UNIT-CHAR = FIRST-UNIT-CHAR / S / HYPHEN / DIGIT

ZERO = "0"

NON-ZERO-DIGIT = ("1"..."9")

DIGIT = ZERO / NON-ZERO-DIGIT

WS = ( S / TAB / NL )

S = U+0020          ; " " (space)

TAB = U+0009        ; "\t" (tab)

NL = U+000A         ; "\n" (newline, linefeed)

HYPHEN = U+000A     ; "-" (hyphen, minus)
```

The ABNF rules `UPPERALPHA`, `LOWERALPHA`, `UNDERSCORE`, `UCS0080TOFFEF` are defined in ANNEX A.

For example, a speedometer may be modeled so that the unit of measure is kilometers per hour. It is necessary to express the derived unit of measure "kilometers per hour" in terms of the allowed base units "meter" and "second". One kilometer per hour is equivalent to

1000 meters per 3600 seconds
or
one meter / second / 3.6

so the programmatic unit for "kilometers per hour" is expressed as: "meter / second / 3.6", using the syntax defined here.

Other examples are as follows:

"meter * meter * 10^-6" → square millimeters
"byte * 2^10" → kBytes as used for memory ("kibobyte")

"byte * 10^3" → kBytes as used for storage ("kilobyte")
"dataword * 4" → QuadWords
"decibel(m) * -1" → -dBm
"second * 250 * 10^-9" → 250 nanoseconds
"foot * foot * foot / minute" → cubic feet per minute, CFM
"revolution / minute" → revolutions per minute, RPM
"pound / inch / inch" → pounds per square inch, PSI
"foot * pound" → foot-pounds

In the "PU Base Unit" column, Table C-1 defines the allowed values for the `base-unit` ABNF rule in the syntax, as well as the empty string indicating no unit. 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 measured by the unit.

The base units in Table C-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 1000:1992. Also, ISO 1000:1992 defines the notational conventions for units, which are used in Table C-1.

**Table C-1 – Base Units for Programmatic Units**

| PU 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)<br>1 dB = 20 · lg (U/U0) | Logarithmic ratio (dimensionless unit)<br>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 |
| 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. |
| MSU | MSU | million service units per hour | A platform-specific, relative measure of the amount of processing work per time performed by a computer, typically used for mainframes. |
| 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) |

| PU Base Unit | Symbol | Calculation | Quantity |
|---|---|---|---|
| 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<br>(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) |
| second | s | SI base unit | Time (duration) |
| minute | min | 1 min = 60 s | Time (duration) |
| hour | h | 1 h = 60 min | Time (duration) |
| day | d | 1 d = 24 h | Time (duration) |
| week | week | 1 week = 7 d | Time (duration) |
| hertz | Hz | 1 Hz = 1 /s | Frequency |
| gravity | g | 1 g = 9.80665 $m/s^2$ | 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 $cd/m^2$ | Luminance |
| lux | lx | 1 lx = 1 $lm/m^2$ | Illuminance |
| newton | N | 1 N = 1 $kg·m/s^2$ | Force |
| pascal | Pa | 1 Pa = 1 $N/m^2$ | 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) |

| PU Base Unit | Symbol | Calculation | Quantity |
|---|---|---|---|
| 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 |
| 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 | 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 |

## C.2 Value for Units Qualifier

**DEPRECATED**

The Units qualifier has been used both for programmatic access and for displaying a unit. Because it does not satisfy the full needs of either of these uses, the Units qualifier is deprecated. The PUnit qualifier should be used instead for programmatic access.

**DEPRECATED**

For displaying a unit, the CIM client should construct the string to be displayed from the PUnit qualifier using the conventions of the CIM client.

The UNITS qualifier specifies the unit of measure in which the qualified property, method return value, or method parameter is expressed. For example, a Size property might have Units (Bytes). The complete set of DMTF-defined values for the Units qualifier is as follows:

- Bits, KiloBits, MegaBits, GigaBits
- < Bits, KiloBits, MegaBits, GigaBits> per Second

- Bytes, KiloBytes, MegaBytes, GigaBytes, Words, DoubleWords, QuadWords
- Degrees C, Tenths of Degrees C, Hundredths of Degrees C, Degrees F, Tenths of Degrees F, Hundredths of Degrees F, Degrees K, Tenths of Degrees K, Hundredths of Degrees K, Color Temperature
- Volts, MilliVolts, Tenths of MilliVolts, Amps, MilliAmps, Tenths of MilliAmps, Watts, MilliWattHours
- Joules, Coulombs, Newtons
- Lumen, Lux, Candelas
- Pounds, Pounds per Square Inch
- Cycles, Revolutions, Revolutions per Minute, Revolutions per Second
- Minutes, Seconds, Tenths of Seconds, Hundredths of Seconds, MicroSeconds, MilliSeconds, NanoSeconds
- Hours, Days, Weeks
- Hertz, MegaHertz
- Pixels, Pixels per Inch
- Counts per Inch
- Percent, Tenths of Percent, Hundredths of Percent, Thousandths
- Meters, Centimeters, Millimeters, Cubic Meters, Cubic Centimeters, Cubic Millimeters
- Inches, Feet, Cubic Inches, Cubic Feet, Ounces, Liters, Fluid Ounces
- Radians, Steradians, Degrees
- Gravities, Pounds, Foot-Pounds
- Gauss, Gilberts, Henrys, MilliHenrys, Farads, MilliFarads, MicroFarads, PicoFarads
- Ohms, Siemens
- Moles, Becquerels, Parts per Million
- Decibels, Tenths of Decibels
- Grays, Sieverts
- MilliWatts
- DBm
- <Bytes, KiloBytes, MegaBytes, GigaBytes> per Second
- BTU per Hour
- PCI clock cycles
- <Numeric value> <Minutes, Seconds, Tenths of Seconds, Hundreths of Seconds, MicroSeconds, MilliSeconds, Nanoseconds>
- Us
- Amps at <Numeric Value> Volts
- Clock Ticks
- Packets, per Thousand Packets

NOTE: Documents using programmatic units may have a need to require that a unit needs to be a particular unit, but without requiring a particular numerical multiplier. That need can be satisfied by statements like: "The programmatic unit shall be 'meter / second' using any numerical multipliers."

# ANNEX D
# (informative)

# UML Notation

The CIM meta-schema notation is directly based on the notation used in Unified Modeling Language (UML). There are distinct symbols for all the major constructs in the schema except qualifiers (as opposed to properties, which are directly represented in the diagrams).

In UML, a class is represented by a rectangle. The class name either stands alone in the rectangle or is in the uppermost segment of the rectangle. If present, the segment below the segment with the name contains the properties of the class. If present, a third region contains methods.

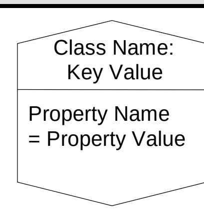
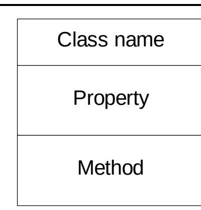
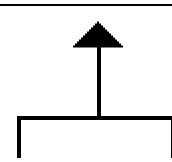
A line decorated with a triangle indicates an inheritance relationship; the lower rectangle represents a subtype of the upper rectangle. The triangle points to the superclass.

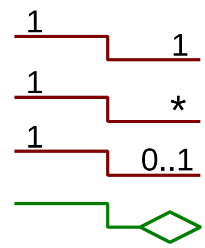
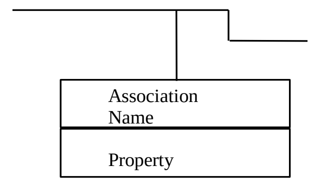
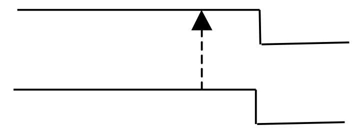
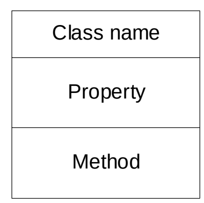
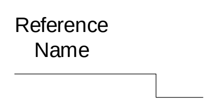
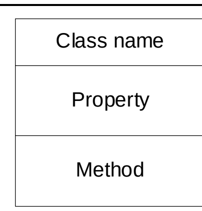
Other solid lines represent relationships. The cardinality of the references on either side of the relationship is indicated by a decoration on either end. The following character combinations are commonly used:

- "1" indicates a single-valued, required reference
- "0…1"  indicates an optional single-valued reference
- "*" indicates an optional many-valued reference (as does "0..*")
- "1..*" indicates a required many-valued reference

A line connected to a rectangle by a dotted line represents a subclass relationship between two associations. The diagramming notation and its interpretation are summarized in Table D-1.

**Table D-1 – Diagramming Notation and Interpretation Summary**

| Meta Element | Interpretation | Diagramming Notation |
|---|---|---|
| Object | |  |
| Primitive type | Text to the right of the colon in the center portion of the class icon | |
| ;Class | |  |
| Subclass | |  |

| Meta Element | Interpretation | Diagramming Notation |
|---|---|---|
| Association | 1:1<br>1:Many<br>1:zero or 1<br>Aggregation |  |
| Association with properties | A link-class that has the same name as the association and uses normal conventions for representing properties and methods |  |
| Association with subclass | A dashed line running from the sub-association to the super class |  |
| Property | Middle section of the class icon is a list of the properties of the class |  |
| Reference | One end of the association line labeled with the name of the reference |  |
| Method | Lower section of the class icon is a list of the methods of the class |  |
| Overriding | No direct equivalent<br>NOTE: Use of the same name does not imply overriding. | |
| Indication | Message trace diagram in which vertical bars represent objects and horizontal lines represent messages | |
| Trigger | State transition diagrams | |
| Qualifier | No direct equivalent | |

# ANNEX E
# (informative)

# Guidelines

The following are general guidelines for CIM modeling:

- Method descriptions are recommended and must, at a minimum, indicate the method’s side effects (pre- and post-conditions).
- Leading underscores in identifiers are to be discouraged and not used at all in the standard schemas.
- It is generally recommended that class names not be reused as part of property or method names. Property and method names are already unique within their defining class.
- To enable information sharing among different CIM implementations, the MaxLen qualifier should be used to specify the maximum length of string properties.
- When extending a schema (i.e., CIM schema or extension schema) with new classes, existing classes should be considered as superclasses of such new classes as appropriate, in order to increase schema consistency.

## E.1 SQL Reserved Words

Avoid using SQL reserved words in class and property names. This restriction particularly applies to property names because class names are prefixed by the schema name, making a clash with a reserved word unlikely. The current set of SQL reserved words is as follows:

From sql1992.txt:

| | | | |
|---|---|---|---|
| AFTER | ALIAS | ASYNC | BEFORE |
| BOOLEAN | BREADTH | COMPLETION | CALL |
| CYCLE | DATA | DEPTH | DICTIONARY |
| EACH | ELSEIF | EQUALS | GENERAL |
| IF | IGNORE | LEAVE | LESS |
| LIMIT | LOOP | MODIFY | NEW |
| NONE | OBJECT | OFF | OID |
| OLD | OPERATION | OPERATORS | OTHERS |
| PARAMETERS | PENDANT | PREORDER | PRIVATE |
| PROTECTED | RECURSIVE | REF | REFERENCING |
| REPLACE | RESIGNAL | RETURN | RETURNS |
| ROLE | ROUTINE | ROW | SAVEPOINT |
| SEARCH | SENSITIVE | SEQUENCE | SIGNAL |
| SIMILAR | SQLEXCEPTION | SQLWARNING | STRUCTURE |
| TEST | THERE | TRIGGER | TYPE |
| UNDER | VARIABLE | VIRTUAL | VISIBLE |
| WAIT | WHILE | WITHOUT | |

From Annex E of sql1992.txt:

| | | | |
|---|---|---|---|
| ABSOLUTE | ACTION | ADD | ALLOCATE |
| ALTER | ARE | ASSERTION | AT |
| BETWEEN | BIT | BIT_LENGTH | BOTH |
| CASCADE | CASCADED | CASE | CAST |
| CATALOG | CHAR_LENGTH | CHARACTER_LENGTH | COALESCE |

| | | | |
|---|---|---|---|
| COLLATE | COLLATION | COLUMN | CONNECT |
| CONNECTION | CONSTRAINT | CONSTRAINTS | CONVERT |
| CORRESPONDING | CROSS | CURRENT_DATE | CURRENT_TIME |
| CURRENT_TIMESTAMP | CURRENT_USER | DATE | DAY |
| DEALLOCATE | DEFERRABLE | DEFERRED | DESCRIBE |
| DESCRIPTOR | DIAGNOSTICS | DISCONNECT | DOMAIN |
| DROP | ELSE | END-EXEC | EXCEPT |
| EXCEPTION | EXECUTE | EXTERNAL | EXTRACT |
| FALSE | FIRST | FULL | GET |
| GLOBAL | HOUR | IDENTITY | IMMEDIATE |
| INITIALLY | INNER | INPUT | INSENSITIVE |
| INTERSECT | INTERVAL | ISOLATION | JOIN |
| LAST | LEADING | LEFT | LEVEL |
| LOCAL | LOWER | MATCH | MINUTE |
| MONTH | NAMES | NATIONAL | NATURAL |
| NCHAR | NEXT | NO | NULLIF |
| OCTET_LENGTH | ONLY | OUTER | OUTPUT |
| OVERLAPS | PAD | PARTIAL | POSITION |
| PREPARE | PRESERVE | PRIOR | READ |
| RELATIVE | RESTRICT | REVOKE | RIGHT |
| ROWS | SCROLL | SECOND | SESSION |
| SESSION_USER | SIZE | SPACE | SQLSTATE |
| SUBSTRING | SYSTEM_USER | TEMPORARY | THEN |
| TIME | TIMESTAMP | TIMEZONE_HOUR | TIMEZONE_MINUTE |
| TRAILING | TRANSACTION | TRANSLATE | TRANSLATION |
| TRIM | TRUE | UNKNOWN | UPPER |
| USAGE | USING | VALUE | VARCHAR |
| VARYING | WHEN | WRITE | YEAR |
| ZONE | | | |

From Annex E of sql3part2.txt:

| | | | |
|---|---|---|---|
| ACTION | ACTOR | AFTER | ALIAS |
| ASYNC | ATTRIBUTES | BEFORE | BOOLEAN |
| BREADTH | COMPLETION | CURRENT_PATH | CYCLE |
| DATA | DEPTH | DESTROY | DICTIONARY |
| EACH | ELEMENT | ELSEIF | EQUALS |
| FACTOR | GENERAL | HOLD | IGNORE |
| INSTEAD | LESS | LIMIT | LIST |
| MODIFY | NEW | NEW_TABLE | NO |
| NONE | OFF | OID | OLD |
| OLD_TABLE | OPERATION | OPERATOR | OPERATORS |
| PARAMETERS | PATH | PENDANT | POSTFIX |
| PREFIX | PREORDER | PRIVATE | PROTECTED |
| RECURSIVE | REFERENCING | REPLACE | ROLE |
| ROUTINE | ROW | SAVEPOINT | SEARCH |
| SENSITIVE | SEQUENCE | SESSION | SIMILAR |
| SPACE | SQLEXCEPTION | SQLWARNING | START |
| STATE | STRUCTURE | SYMBOL | TERM |
| TEST | THERE | TRIGGER | TYPE |
| UNDER | VARIABLE | VIRTUAL | VISIBLE |
| WAIT | WITHOUT | | |

From Annex E of sql3part4.txt:

| | | | |
|---|---|---|---|
| CALL | DO | ELSEIF | EXCEPTION |
| IF | LEAVE | LOOP | OTHERS |
| RESIGNAL | RETURN | RETURNS | SIGNAL |
| TUPLE | WHILE | | |

# ANNEX F
# (normative)

# EmbeddedObject and EmbeddedInstance Qualifiers

Use of the EmbeddedObject and EmbeddedInstance qualifiers is motivated by the need to include the data of a specific instance in an indication (event notification) or to capture the contents of an instance at a point in time (for example, to include the CIM_DiagnosticSetting properties that dictate a particular CIM_DiagnosticResult in the Result object).

Therefore, the next major version of the CIM Specification is expected to include a separate data type for directly representing instances (or snapshots of instances). Until then, the EmbeddedObject and EmbeddedInstance qualifiers can be used to achieve an approximately equivalent effect. They permit a CIM object manager (or other entity) to simulate embedded instances or classes by encoding them as strings when they are presented externally. Embedded instances can have properties that again are defined to contain embedded objects. CIM clients that do not handle embedded objects may treat properties with this qualifier just like any other string-valued property. CIM clients that do want to realize the capability of embedded objects can extract the embedded object information by decoding the presented string value.

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 containing instance. This dependency makes the string value appear to vary according to the circumstances in which it is observed. This is an acknowledged weakness of using a qualifier instead of a new data type.

This document defines the encoding of embedded objects for the MOF representation and for the CIM-XML protocol. When other protocols or representations are used to communicate with embedded object-aware consumers of CIM data, they must include particulars on the encoding for the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance.

## F.1 Encoding for MOF

When the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance are rendered in MOF, the embedded object must be encoded into string form using the MOF syntax for the `instanceDeclaration` nonterminal in embedded instances or for the `classDeclaration`, `assocDeclaration`, or `indicDeclaration` ABNF rules, as appropriate in embedded classes (see ANNEX A).

EXAMPLES:

```
instance of CIM_InstCreation {
   EventTime = "20000208165854.457000-360";
   SourceInstance =
      "instance of CIM_Fan {\n"
      "DeviceID = \"Fan 1\";\n"
      "Status = \"Degraded\";\n"
      "};\n";
};

instance of CIM_ClassCreation {
   EventTime = "20031120165854.457000-360";
   ClassDefinition =
      "class CIM_Fan : CIM_CoolingDevice {\n"
```

```
       "   boolean VariableSpeed;\n"
       "      [Units (\"Revolutions per Minute\")]\n"
       "   uint64 DesiredSpeed;\n"
       "};\n"
};
```

## F.2 Encoding for CIM Protocols

The rendering of values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance in CIM protocols is defined in the specifications defining these protocols.

# ANNEX G
# (informative)

# Schema Errata

Based on the concepts and constructs in this document, the CIM schema is expected to evolve for the following reasons:

- To add new classes, associations, qualifiers, properties and/or methods. This task is addressed in 5.4.
- To correct errors in the Final Release versions of the schema. This task fixes errata in the CIM schemas after their final release.
- To deprecate and update the model by labeling classes, associations, qualifiers, and so on as "not recommended for future development" and replacing them with new constructs. This task is addressed by the Deprecated qualifier described in 5.6.3.11.

Examples of errata to correct in CIM schemas are as follows:

- Incorrectly or incompletely defined keys (an array defined as a key property, or incompletely specified propagated keys)
- Invalid subclassing, such as subclassing an optional association from a weak relationship (that is, a mandatory association), subclassing a nonassociation class from an association, or subclassing an association but having different reference names that result in three or more references on an association
- Class references reversed as defined by an association's roles (antecedent/dependent references reversed)
- Use of SQL reserved words as property names
- Violation of semantics, such as missing Min(1) on a Weak relationship, contradicting that a weak relationship is mandatory

Errata are a serious matter because the schema should be correct, but the needs of existing implementations must be taken into account. Therefore, the DMTF has defined the following process (in addition to the normal release process) with respect to any schema errata:

a) Any error should promptly be reported to the Technical Committee (technical@dmtf.org) for review. Suggestions for correcting the error should also be made, if possible.

b) The Technical Committee documents its findings in an email message to the submitter within 21 days. These findings report the Committee's decision about whether the submission is a valid erratum, the reasoning behind the decision, the recommended strategy to correct the error, and whether backward compatibility is possible.

c) If the error is valid, an email message is sent (with the reply to the submitter) to all DMTF members (members@dmtf.org). The message highlights the error, the findings of the Technical Committee, and the strategy to correct the error. In addition, the committee indicates the affected versions of the schema (that is, only the latest or all schemas after a specific version).

d) All members are invited to respond to the Technical Committee within 30 days regarding the impact of the correction strategy on their implementations. The effects should be explained as thoroughly as possible, as well as alternate strategies to correct the error.

e) If one or more members are affected, then the Technical Committee evaluates all proposed alternate correction strategies. It chooses one of the following three options:

   – To stay with the correction strategy proposed in b)

   – To move to one of the proposed alternate strategies

   – To define a new correction strategy based on the evaluation of member impacts

f) If an alternate strategy is proposed in Item e), the Technical Committee may decide to reenter the errata process, resuming with Item c) and send an email message to all DMTF members about the alternate correction strategy. However, if the Technical Committee believes that further comment will not raise any new issues, then the outcome of Item e) is declared to be final.

g) If a final strategy is decided, this strategy is implemented through a Change Request to the affected schema(s). The Technical Committee writes and issues the Change Request. Affected models and MOF are updated, and their introductory comment section is flagged to indicate that a correction has been applied.

# ANNEX H
# (informative)

# Ambiguous Property and Method Names

In 5.1.2.8 it is explicitly allowed for a subclass to define a property that may have the same name as a property defined by a superclass and for that new property not to override the superclass property. The subclass may override the superclass property by attaching an Override qualifier; this situation is well-behaved and is not part of the problem under discussion.

Similarly, a subclass may define a method with the same name as a method defined by a superclass without overriding the superclass method. This annex refers only to properties, but it is to be understood that the issues regarding methods are essentially the same. For any statement about properties, a similar statement about methods can be inferred.

This same-name capability allows one group (the DMTF, in particular) to enhance or extend the superclass in a minor schema change without to coordinate with, or even to know about, the development of the subclass in another schema by another group. That is, a subclass defined in one version of the superclass should not become invalid if a subsequent version of the superclass introduces a new property with the same name as a property defined on the subclass. Any other use of the same-name capability is strongly discouraged, and additional constraints on allowable cases may well be added in future versions of CIM.

It is natural for CIM clients to be written under the assumption that property names alone suffice to identify properties uniquely. However, such CIM clients risk failure if they refer to properties from a subclass whose superclass has been modified to include a new property with the same name as a previously-existing property defined by the subclass.

For example, consider the following:

```
   [Abstract]
class CIM_Superclass
{
};
```

```
class VENDOR_Subclass
{
   string Foo;
};
```

Assuming CIM-XML as the CIM protocol and assuming only one instance of VENDOR_Subclass, invoking the EnumerateInstances operation on the class "VENDOR_Subclass" without also asking for class origin information might produce the following result:

```
<INSTANCE CLASSNAME="VENDOR_Subclass">
   <PROPERTY NAME="Foo" TYPE="string">
      <VALUE>Hello, my name is Foo</VALUE>
   </PROPERTY>
</INSTANCE>
```

If the definition of CIM_Superclass changes to:

```
   [Abstract]
class CIM_Superclass
```

```
{
   string Foo = "You lose!";
};
```

then the EnumerateInstances operation might return the following:

```
<INSTANCE>
   <PROPERTY NAME="Foo" TYPE="string">
      <VALUE>You lose!</VALUE>
   </PROPERTY>
   <PROPERTY NAME="Foo" TYPE="string">
      <VALUE>Hello, my name is Foo</VALUE>
   </PROPERTY>
</INSTANCE>
```

If the CIM client attempts to retrieve the 'Foo' property, the value it obtains (if it does not experience an error) depends on the implementation.

Although a class may define a property with the same name as an inherited property, it may not define two (or more) properties with the same name. Therefore, the combination of defining class plus property name uniquely identifies a property. (Most CIM operations that return instances have a flag controlling whether to include the class origin for each property. For example, in DSP0200, see the clause on EnumerateInstances; in DSP0201, see the clause on ClassOrigin.)

However, the use of class-plus-property-name for identifying properties makes a CIM client vulnerable to failure if a property is promoted to a superclass in a subsequent schema release. For example, consider the following:

```
class CIM_Top
{
};

class CIM_Middle : CIM_Top
{
   uint32 Foo;
};

class VENDOR_Bottom : CIM_Middle
{
   string Foo;
};
```

A CIM client that identifies the uint32 property as "the property named 'Foo' defined by CIM_Middle" no longer works if a subsequent release of the CIM schema changes the hierarchy as follows:

```
class CIM_Top
{
   uint32 Foo;
};

class CIM_Middle : CIM_Top
{
};

```

```
class VENDOR_Bottom : CIM_Middle
{
   string Foo;
};
```

Strictly speaking, there is no longer a "property named 'Foo' defined by CIM_Middle"; it is now defined by CIM_Top and merely inherited by CIM_Middle, just as it is inherited by VENDOR_Bottom. An instance of VENDOR_Bottom returned in CIM-XML from a CIM server might look like this:

```
<INSTANCE CLASSNAME="VENDOR_Bottom">
   <PROPERTY NAME="Foo" TYPE="string" CLASSORIGIN="VENDOR_Bottom">
      <VALUE>Hello, my name is Foo!</VALUE>
   </PROPERTY>
   <PROPERTY NAME="Foo" TYPE="uint32" CLASSORIGIN="CIM_Top">
      <VALUE>47</VALUE>
   </PROPERTY>
</INSTANCE>
```

A CIM client looking for a PROPERTY element with NAME="Foo" and CLASSORIGIN="CIM_Middle" fails with this XML fragment.

Although CIM_Middle no longer defines a 'Foo' property directly in this example, we intuit that we should be able to point to the CIM_Middle class and locate the 'Foo' property that is defined in its nearest superclass. Generally, a CIM client must be prepared to perform this search, separately obtaining information, when necessary, about the (current) class hierarchy and implementing an algorithm to select the appropriate property information from the instance information returned from a CIM operation.

Although it is technically allowed, schema writers should not introduce properties that cause name collisions within the schema, and they are strongly discouraged from introducing properties with names known to conflict with property names of any subclass or superclass in another schema.

# ANNEX I
# (informative)

# OCL Considerations

The Object Constraint Language (OCL) is a formal language to describe expressions on models. It is defined by the Open Management Group (OMG) in the *Object Constraint Language* specification, which describes OCL as follows:

- OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition).
- OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, OCL expressions are not by definition directly executable.
- OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set of supplementary predefined types. These are described in Chapter 11 ("The OCL Standard Library").
- As a specification language, all implementation issues are out of scope and cannot be expressed in OCL. The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation."

For a particular CIM class, more than one CIM association referencing that class with one reference can define the same name for the opposite reference. OCL allows navigation from an instance of such a class to the instances at the other end of an association using the name of the opposite association end (that is, a CIM reference). However, in the case discussed, that name is not unique. For OCL statements to tolerate the future addition of associations that create such ambiguity, OCL navigation from an instance to any associated instances should first navigate to the association class and from there to the associated class, as described in the *Object Constraint Language* specification in its sections 7.5.4 "Navigation to Association Classes" and 7.5.5 "Navigation from Association Classes". OCL requires the first letter of the association class name to be lowercase when used for navigating to it. For example, CIM_Dependency becomes cIM_Dependency.

EXAMPLE:

```
   [ClassConstraint {
    "inv i1: self.p1 = self.acme_A12.r.p2"}]
       // Using class name ACME_A12 is required to disambiguate end name r
class ACME_C1 {
   string p1;
};

   [ClassConstraint {
    "inv i2: self.p2 = self.acme_A12.x.p1",  // Using ACME_A12 is recommended
    "inv i3: self.p2 = self.x.p1"}]          // Works, but not recommended
class ACME_C2 {
   string p2;
};
```

```

class ACME_C3 { };

   [Association]
class ACME_A12 {
   ACME_C1 REF x;
   ACME_C2 REF r;  // same name as ACME_A13::r
};

   [Association]
class ACME_A13 {
   ACME_C1 REF y;
   ACME_C3 REF r;  // same name as ACME_A12::r
};
```

# ANNEX J
# (informative)

# Change Log

| Version | Date | Description |
|---|---|---|
| 1 | 1997-04-09 | First Public Release |
| 2.2 | 1999-06-14 | Released as Final Standard |
| 2.2.1000 | 2003-06-07 | Released as Final Standard |
| 2.3 | 2004-08-11 | Released as Preliminary Standard |
| 2.3 | 2005-10-04 | Released as Final Standard |
| 2.4.0a | 2007-11-12 | Released as Preliminary Standard |
| 2.5.0a | 2008-04-22 | Released as Preliminary Standard |
| 2.5.0 | 2009-03-04 | Released as DMTF Standard |
| 2.6.0a | 2009-11-04 | Released as a Work in Progress |
| 2.6.0 | 2010-03-17 | Released as DMTF Standard |
| 2.7.0 | 2012-04-22 | Released as DMTF Standard, with the following changes since version 2.6.0:<br>• Deprecated allowing class as object reference in method parameters<br>• Added Reference qualifier (Mantis 1116, ARCHCR00142)<br>• Added Structure qualifier<br>• Removed class from scope of Exception qualifier<br>• Added programmatic unit "MSU" (Mantis 0679)<br>• Clarified timezone ambiguities in timestamps (Mantis 1165)<br>• Fixed incorrect mixup of property default value and initialization constraint (Mantis 1146)<br>• Defined backward compatibility between client, server and listener.<br>• Clarified ambiguities related to initialization constraints (Mantis 0925)<br>• Fixed outdated & incorrect statements in "CIM Implementation Conformance" (Mantis 0681)<br>• Fixed inconsistent language in description of Null (Mantis 1065)<br>• Fixed incorrect use of normative language in ModelCorrespondence example (Mantis 0900)<br>• Removed policy example<br>• Clarified use of term "top-level" (Mantis 1050)<br>• Added term for "UCS character" (Mantis 1082)<br>• Added term for the combined unit in programmatic units (Mantis 0680)<br>• Fixed inconsistenties in lexical case for TRUE, FALSE, NULL (Mantis 0821)<br>• Small editorial issues (Mantis 0820)<br>• Added folks to list of contributors |

# Bibliography

Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Document Set*, Rational Software Corporation, 1996, http://www.rational.com/uml

James O. Coplein, Douglas C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, Reading Mass., 1995

Georges Gardarin and Patrick Valduriez, *Relational Databases and Knowledge Bases*, Addison Wesley, 1989

Gerald M. Weinberg, *An Introduction to General Systems Thinking*, 1975 ed. Wiley-Interscience, 2001 ed. Dorset House

DMTF DSP0200, *CIM Operations over HTTP*, Version 1.3 http://www.dmtf.org/standards/published_documents/DSP0200_1.3.pdf

DMTF DSP0201, *Specification for the Representation of CIM in XML*, Version 2.3 http://www.dmtf.org/standards/published_documents/DSP0201_2.3.pdf

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

IETF, RFC2068, *Hypertext Transfer Protocol – HTTP/1.1*, http://tools.ietf.org/html/rfc2068

IETF, RFC1155, *Structure and Identification of Management Information for TCP/IP-based Internets*, http://tools.ietf.org/html/rfc1155

IETF, RFC2253, *Lightweight Directory Access Protocol (v3): UTF-8 String Representation Of Distinguished Names*, http://tools.ietf.org/html/rfc2253

OMG, *Unified Modeling Language: Infrastructure*, Version 2.1.1 http://www.omg.org/cgi-bin/doc?formal/07-02-06

The Unicode Consortium: *The Unicode Standard*, http://www.unicode.org

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

W3C, *XML Schema Part 0: Primer Second Edition*, W3C Recommendation, 28 October 2004, http://www.w3.org/TR/xmlschema-0/