Document Number: DSP0004

Date: 2009-05-01

Version: 2.5.0

Common Information Model (CIM) Infrastructure

Document Type: Specification

Document Status: DMTF Standard

Document Language: E

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.

Throughout this document, elements of formal syntax are described in the notation defined in RFC 4234, with these deviations:

- Each token may be separated by an arbitrary number of white space characters unless otherwise stated (at least one tab, carriage return, line feed, form feed, or space).
- The vertical bar `("|")` character is used to express alternation rather than the virgule `("/")` specified in RFC 4234.

The DMTF acknowledges the following people.

Editor:

- Lawrence Lamers – VMware

Contributors:

- Jeff Piazza – HP
- Andreas Maier – IBM
- George Ericson – EMC
- Jim Davis – WBEM Solutions
- Karl Schopmeyer – Inova Development
- Steve Hand – Symantec

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.

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.

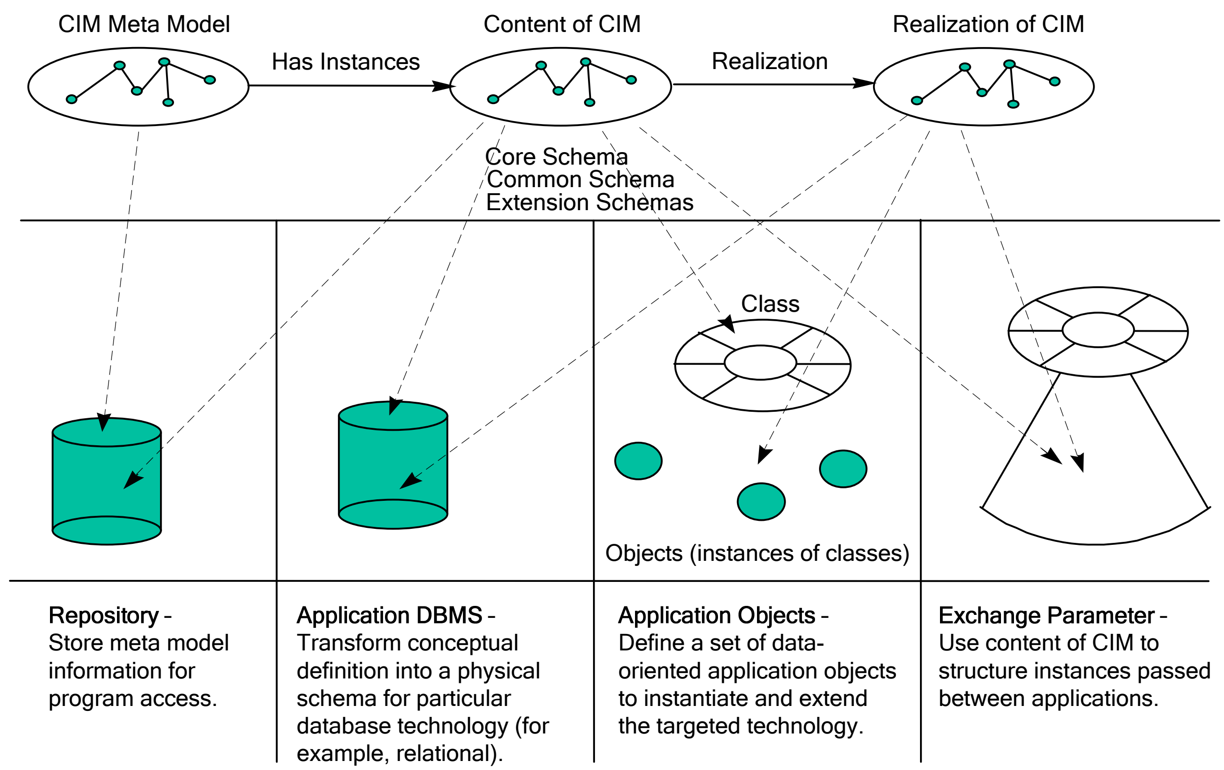


Figure 1 – Four Ways to Use CIM

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

The ability to exchange information between management applications is fundamental to CIM. The current exchange mechanism is the Managed Object Format (MOF). As of now,[1] no programming interfaces or protocols are defined by (and thus cannot be considered as) an exchange mechanism. Therefore, a CIM-capable system must be able to import and export properly formed MOF constructs. How the import and export operations are performed is an implementation detail for the CIM-capable system.

Objects instantiated in the MOF must, at a minimum, include all key properties and all required properties. Required properties have the Required qualifier present and are set to TRUE.

Trademarks

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

[1] The standard CIM application programming interface and/or communication protocol will be defined in a future version of the CIM Infrastructure 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 references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Copies of the following documents may be obtained from ANSI:

a) approved ANSI standards;

b) approved and draft international and regional standards (e.g., ISO, IEC); and

c) approved and draft foreign standards (e.g., JIS and DIN).

For further information, contact ANSI Customer Service Department at 212-642-4900 (phone), 212-302-1286 (fax) or via the World Wide Web at http://www.ansi.org.

Additional availability contact information is provided below as needed.

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
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

2.1 Approved References

ANSI/IEEE Standard 754-1985, *IEEE® Standard for Binary Floating-Point Arithmetic*, Institute of Electrical and Electronics Engineers, August 1985.

CCITT X.680 (07/02) *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*

DMTF DSP0200, *CIM Operations over HTTP*, Version 1.3

DMTF DSP4004, *DMTF Release Process*, Version 2.1

DMTF DSP0201, *Specification for the Representation of CIM in XML*, Version 2.3

EIA-310 Cabinets, Racks, Panels, and Associated Equipment

ISO 639-1:2002 *Codes for the representation of names of languages – Part 1: Alpha-2 code*

ISO 639-2:1998 *Codes for the representation of names of languages – Part 2: Alpha-3 code*

ISO 639-3:2007 *Codes for the representation of names of languages – Part 3: Alpha-3 code for comprehensive coverage of languages*

ISO 1000:1992 *SI units and recommendations for the use of their multiples and of certain other units*

ISO 3166-1:2006 *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*

ISO 3166-2:2007 *Codes for the representation of names of countries and their subdivisions – Part 2: Country subdivision code*

ISO 3166-3:1999 *Codes for the representation of names of countries and their subdivisions – Part 3: Code for formerly used names of countries*

ISO 8601:2004 (E), *Data elements and interchange formats – Information interchange – Representation of dates and times*

ISO/IEC 9075-10:2003 *Information technology – Database languages – SQL – Part 10: Object Language Bindings (SQL/OLB)*

ISO/IEC 10165-4:1992 Information technology – Open Systems Interconnection – Structure of management information – Part 4: Guidelines for the definition of managed objects (GDMO)

ISO/IEC 10646:2003 *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*

ISO/IEC 14750:1999 *Information technology – Open Distributed Processing – Interface Definition Language*

ITU X.501: *Information Technology – Open Systems Interconnection – The Directory: Models*

OMG, *Object Constraint Language Version 2.0*

OMG, *UML Superstructure Specification, Version 2.1.1*

OMG, *UML Infrastructure Specification, Version 2.1.1*

OMG, *UML OCL Specification, Version 2.0*

2.2 Other References

ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*

IETF, RFC 2068, *Hypertext Transfer Protocol – HTTP/1.1*

IETF, RFC 1155, *Structure and Identification of Management Information for TCP/IP-based Internets*

IETF, RFC 2253, *Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names*

IETF, RFC 2279, *UTF-8, a transformation format of ISO 10646*

IETF, RFC 4234, *Augmented BNF for Syntax Specifications: ABNF*, 2005

3 Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

The keywords can, cannot, shall, shall not, should, should not, may, and may not in this document are to be interpreted as described in ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards.*

3.1 Keywords

3.1.1
conditional
indicates requirements to be followed strictly in order to conform to the document when the specified conditions are met

3.1.2
mandatory
indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.1.3
optional
indicates a course of action permissible within the limits of the document

3.1.4
unspecified
indicates that this profile does not define any constraints for the referenced CIM element or operation

3.2 Terms

3.2.1
aggregation
A strong form of an *association*. For example, the containment relationship between a system and its components can be called an *aggregation*. An *aggregation* is expressed as a *qualifier* on the *association* class. *Aggregation* often implies, but does not require, the aggregated *objects* to have mutual dependencies.

3.2.2
association

A *class* that expresses the relationship between two other *classes*. The relationship is established by two or more *references* in the *association class* pointing to the related *classes*.

3.2.3
cardinality

A relationship between two classes that allows more than one *object* to be related to a single *object*. For example, Microsoft Office* is made up of the software elements Word, Excel, Access, and PowerPoint.

3.2.4
Common Information Model
CIM

Common Information Model is the schema of the overall managed environment. It is divided into a *core model*, *common model,* and *extended schemas*.

3.2.5
CIM schema

The schema representing the *core* and *common models*. The DMTF releases versions of this schema over time as the schema evolves.

3.2.6
class

A collection of instances that all support a common type; that is, a set of *properties* and *methods*. The common *properties* and *methods* are defined as *features* of the *class*. For example, the *class* called Modem represents all the modems present in a system.

3.2.7
common model

A collection of *models* specific to a particular area and derived from the *core model*. Included are the system *model*, the application *model*, the network *model,* and the device *model*.

3.2.8
core model

A subset of CIM that is not specific to any platform. The *core model* is set of *classes* and *associations* that establish a conceptual framework for the *schema* of the rest of the managed environment. Systems, applications, networks, and related information are modeled as extensions to the *core model*.

3.2.9
domain

A virtual room for object names that establishes the range in which the names of objects are unique.

3.2.10
explicit qualifier

A *qualifier* defined separately from the definition of a *class*, *property,* or other schema element (see *implicit qualifier*). *Explicit qualifier* names shall be unique across the entire *schema*. *Implicit qualifier* names shall be unique within the defining schema element; that is, a given schema element shall not have two *qualifiers* with the same name.

3.2.11
extended schema

A platform-specific *schema* derived from the common model. An example is the Win32 *schema*.

3.2.12

feature

A *property* or *method* belonging to a *class*.

3.2.13

flavor

Part of a *qualifier* specification indicating overriding and *inheritance* rules. For example, the *qualifier* KEY has Flavor(DisableOverride ToSubclass), meaning that every subclass must inherit it and cannot override it.

3.2.14

implicit qualifier

A *qualifier* that is a part of the definition of a *class*, *property,* or other schema element (see *explicit qualifier*).

3.2.15

indication

A type of *class* usually created as a result of a *trigger*.

3.2.16

inheritance

A relationship between two *classes* in which all members of the *subclass* are required to be members of the *superclass*. Any member of the *subclass* must also support any *method* or *property* supported by the *superclass*. For example, Modem is a *subclass* of Device.

3.2.17

instance

A unit of data. An *instance* is a set of *property* values that can be uniquely identified by a *key*.

3.2.18

key

One or more qualified class properties that can be used to construct a name.

One or more qualified object properties that uniquely identify instances of this object in a namespace.

3.2.19

managed object

The actual item in the system environment that is accessed by the *provider* — for example, a network interface card.

3.2.20

meta model

A set of *classes*, *associations,* and *properties* that expresses the types of things that can be defined in a *Schema*. For example, the *meta model* includes a *class* called property that defines the *properties* known to the system, a *class* called method that defines the *methods* known to the system, and a *class* called class that defines the *classes* known to the system.

3.2.21

meta schema

The schema of the meta model.

3.2.22

method

A declaration of a signature, which includes the method name, return type, and parameters. For a concrete class, it may imply an implementation.

3.2.23

model

A set of *classes*, *associations,* and *properties* that allows the expression of information about a specific domain. For example, a network may consist of network devices and logical networks. The network devices may have attachment *associations* to each other, and they may have member *associations* to logical networks.

3.2.24

model path

A reference to an object within a namespace.

3.2.25

namespace

An *object* that defines a scope within which object keys must be unique.

3.2.26

namespace path

A reference to a namespace within an implementation that can host CIM objects.

3.2.27

name

The combination of a namespace path and a model path that identifies a unique object.

3.2.28

polymorphism

A *subclass* may redefine the implementation of a *method* or *property* inherited from its *superclass*. The *property* or *method* is therefore redefined, even if the *superclass* is used to access the object. For example, Device may define availability as a string, and may return the values "powersave," "on," or "off." The Modem *subclass* of Device may redefine (override) availability by returning "on" or "off," but not "powersave". If all Devices are enumerated, any Device that happens to be a modem does not return the value "powersave" for the availability *property*.

3.2.29

property

A value used to characterize an instance of a *class*. For example, a Device may have a *property* called status.

3.2.30

provider

An executable that can return or set information about a given *managed object*.

3.2.31

qualifier

A value used to characterize a *method*, *property*, or *class* in the *meta schema*. For example, if a property has the Key qualifier with the value TRUE, the property is a key for the class.

3.2.32

reference

Special *property types* that are references or pointers to other instances.

3.2.33

schema

A management schema is provided to establish a common conceptual framework at the level of a fundamental topology both for classification and association and for a basic set of classes to establish a common framework to describe the managed environment. A *schema* is a namespace and unit of ownership for a set of classes. *Schemas* may take forms such as a text file, information in a repository, or diagrams in a CASE tool.

3.2.34

scope

Part of a *qualifier* specification indicating the meta constructs with which the *qualifier* can be used. For example, the Abstract *qualifier* has Scope(Class Association Indication), meaning that it can be used only with *classes*, *associations*, and *indications*.

3.2.35

scoping object

An object that represents a real-world managed element, which in turn propagates keys to other objects.

3.2.36

signature

The return type and parameters supported by a *method*.

3.2.37

subclass

See inheritance.

3.2.38

superclass

See inheritance.

3.2.39

top-level object
(TLO)

A class or object that has no scoping object.

3.2.40

trigger

The occurrence of some action such as the creation, modification, or deletion of an *object*, access to an *object,* or modification or access to a *property*. *Triggers* may also be fired when a specified period of time passes. A *trigger* typically results in an *indication*.

4 Symbols and Abbreviated Terms

The following symbols and 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

TLO

top-level object

4.14

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) 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 elements of the model are schemas, classes, properties, and methods. The model also supports indications and associations as types of classes and references as types of properties. The elements of the model are described in the following list:

- *Schema*

  A group of classes with a single owner. Schemas are used for administration and class naming. Class names must be unique within their schemas.

- *Class*

  A collection of instances that support the same type (that is, the same properties and methods).

  Classes can 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. Classes can have methods, which represent their behavior. A class can participate in associations as the target of a reference owned by the association. Classes also have instances (not represented in Figure 2).

- *Instance*

  Each instance provides values for the properties associated with its defining Class. An instance does not carry values for any other properties or methods not defined in (or inherited by) its defining class. An instance cannot redefine the properties or methods defined in (or inherited by) its defining class.

  Instances are not named elements and cannot have qualifiers associated with them. However, qualifiers may be associated with the instance's class, as well as with the properties and methods defined in or inherited by that class. Instances cannot attach new qualifiers to properties, methods, or parameters because the association between qualifier and named element is not restricted to the context of a particular instance.

- *Property*

  Assigns values to characterize instances of a class. A property can be thought of as a pair of Get and Set functions that return state and set state, respectively, when they are applied to an object.[2]

- *Method*

  A declaration of a signature (that is, the method name, return type, and parameters). For a concrete class, it may imply an implementation.

  Properties and methods have reflexive associations that represent property and method overriding. A method can override an inherited method so that any access to the inherited method invokes the implementation of the overriding method. Properties are overridden in the same way.

- *Trigger*

  Recognition of a state change (such as create, delete, update, or access) of a class instance, and update of or access to a property.

---

[2] Note the equivocation between "object" as instance and "object" as class. This is common usage in object-oriented literature and reflects the fact that, in many cases, operations and concepts may apply to or involve both classes and instances.

- *Indication*

  An object created as a result of a trigger. Because indications are subtypes of a class, they can have properties and methods and they can be arranged in a type hierarchy.

- *Association*

  A class that contains two or more references. An association represents a relationship between two or more objects. A relationship can be established between classes without affecting any related classes. That is, an added association does not affect the interface of the related classes. Associations have no other significance. Only associations can have references. An association cannot be a subclass of a non-association class. Any subclass of an association is an association.

- *Reference*

  Defines the role each object plays in an association. The reference represents the role name of a class in the context of an association. A given object can have multiple relationship instances. For example, a system can be related to many system components.

- *Qualifier*

  Characterizes named elements. For example, qualifiers can define the characteristics of a property or the key of a class. Specifically, qualifiers can characterize classes (including associations and indications), properties (including references), methods, and method parameters. Qualifiers do not characterize qualifier types and do not characterize other qualifiers. Qualifiers make the meta schema extensible in a limited and controlled fashion. New types of qualifiers can be added by introducing a new qualifier name, thereby providing new types of meta data to processes that manage and manipulate classes, properties, and other elements of the meta schema.

Figure 2 provides an overview of the structure of the meta schema. The complete meta schema is defined by the MOF in ANNEX B. The rules defining the meta schema are as follows:

1) Every meta construct is expressed as a descendent of a named element.
2) A named element has zero or more characteristics. A characteristic is a qualifier for a named element.
3) A named element can trigger zero or more indications.
4) A schema is a named element and can contain zero or more classes. A class must belong to only one schema.
5) A qualifier type (not shown in Figure 2) is a named element and must supply a type for a qualifier (that is, a qualifier must have a qualifier type). A qualifier type can be used to type zero or more qualifiers.

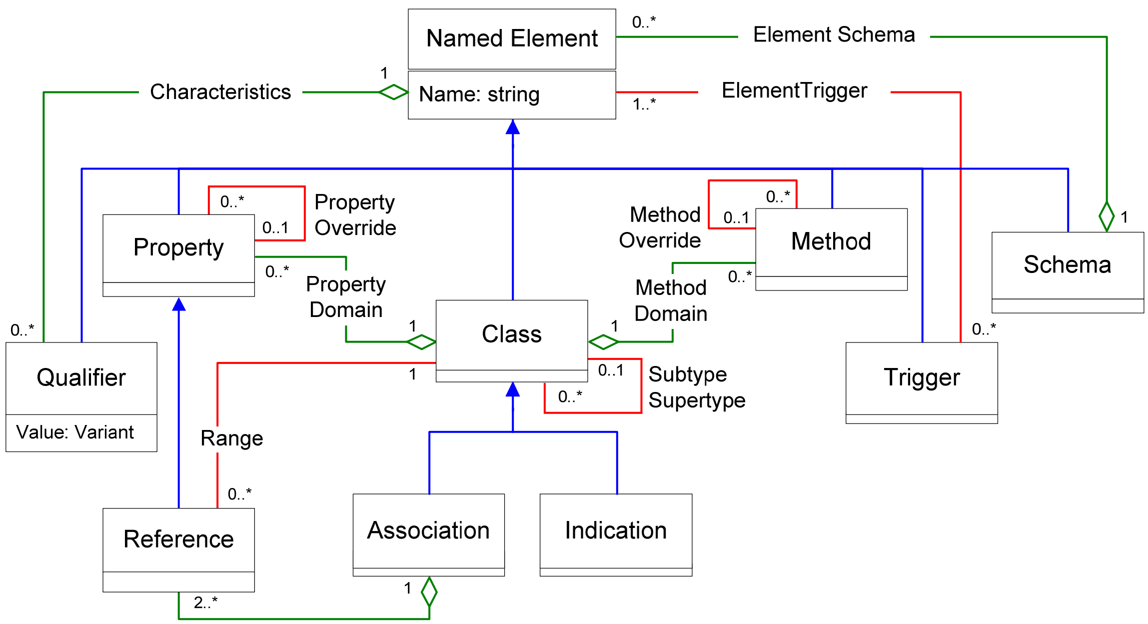


Figure 2 – Meta Schema Structure

6) A qualifier is a named element and has a name, a type (intrinsic data type), a value of this type, a scope, a flavor, and a default value. The type of the qualifier value must agree with the type of the qualifier type.

7) A property is a named element with exactly one domain: the class that owns the property. The property can apply to instances of the domain (including instances of subclasses of the domain) and not to any other instances.

8) A property can override another property from a different class. The domain of the overridden property must be a supertype of the domain of the overriding property. For non-reference properties, the type of the overriding property shall be the same as the type of the overridden property. For References, the range of the overriding Reference shall be the same as, or a subclass of, the range of the overridden Reference.

9) The class referenced by the range association (Figure 5) of an overriding reference must be the same as, or a subtype of, the class referenced by the range associations of the overridden reference.

10) The domain of a reference must be an association.

11) A class is a type of named element. A class can have instances (not shown on the diagram) and is the domain for zero or more properties. A class is the domain for zero or more methods.

12) A class can have zero or one supertype and zero or more subtypes.

13) An association is a type of class. Associations are classes with an association qualifier.

14) An association must have two or more references.

15) An association cannot inherit from a non-association class.

16) Any subclass of an association is an association.

17) A method is a named element with exactly one domain: the class that owns the method. The method can apply to instances of the domain (including instances of subclasses of the domain) and not to any other instances.

18) A method can override another method from a different class. The domain of the overridden method must be a superclass of the domain of the overriding method.

19) A trigger is an operation that is invoked on any state change, such as object creation, deletion, modification, or access, or on property modification or access. Qualifiers, qualifier types, and schemas may not have triggers. The changes that invoke a trigger are specified as a qualifier.

20) An indication is a type of class and has an association with zero or more named triggers that can create instances of the indication.

21) Every meta-schema object is a descendent of a named element. All names are case-insensitive. The naming rules, which vary depending on the creation type of the object, are as follows:

    a) Fully-qualified class names (that is, prefixed by the schema name) are unique within the schema.

    b) Fully-qualified association and indication names are unique within the schema (implied by the fact that associations and indications are subtypes of class).

    c) Implicitly-defined qualifier names are unique within the scope of the characterized object. That is, a named element may not have two characteristics with the same name. Explicitly-defined qualifier names are unique within the defining namespace and must agree in type, scope, and flavor with any explicitly-defined qualifier of the same name.

    d) Trigger names must be unique within the property, class, or method to which they apply.

    e) Method and property names must be unique within the domain class. A class can inherit more than one property or method with the same name. Property and method names can be qualified using the name of the declaring class.

    f) Reference names must be unique within the scope of their defining association and obey the same rules as property names. Reference names do not have to be unique within the scope of the related class because the reference provides the name of the class in the context defined by the association (Figure 3).

It is legal for the class system to be related to service by two independent associations (*dependency* and *hosted services,* each with roles *system* and *service*). However, *hosted services* cannot define another reference *service* to the service class because a single association would then contain two references called *service.*

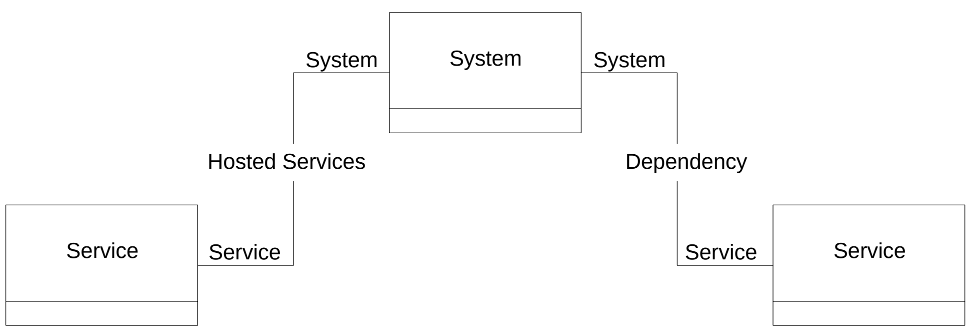


Figure 3 – Reference Naming

22) Qualifiers are characteristics of named elements. A qualifier has a name (inherited from a named element) and a value that defines the characteristics of the named element. For example, a class can have a qualifier named "Description," the value of which is the description for the class. A property can have a qualifier named "Units" that has values such as "bytes" or "kilobytes." The value is a variant (that is, a value plus a type).

23) Association and indication are types of class, so they can be the domain for methods, properties, and references. That is, associations and indications can have properties and methods just as a class does. Associations and indications can have instances. The instance of an association has a set of references that relate one or more objects. An instance of an indication represents an event and is created because of that event — usually a trigger. Indications are not required to have keys. Typically, indications are very short-lived objects to communicate information to an event consumer.

24) A reference has a range that represents the type of the Reference. For example, in the model of PhysicalElements and PhysicalPackages (Figure 4), there are two references:

– ContainedElement has PhysicalElement as its range and container as its domain.

– ContainingElement has PhysicalPackage as its range and container as its domain.

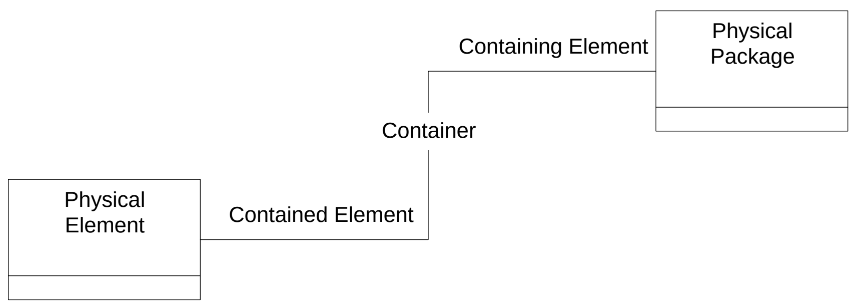


Figure 4 – References, Ranges, and Domains

25) A class has a subtype-supertype association for substitutions so that any instance of a subtype can be substituted for any instance of the supertype in an expression without invalidating the expression.

In the container example (Figure 5), Card is a subtype of PhysicalPackage. Therefore, Card can be used as a value for the ContainingElement reference. That is, an instance of Card can be used as a substitute for an instance of PhysicalPackage.

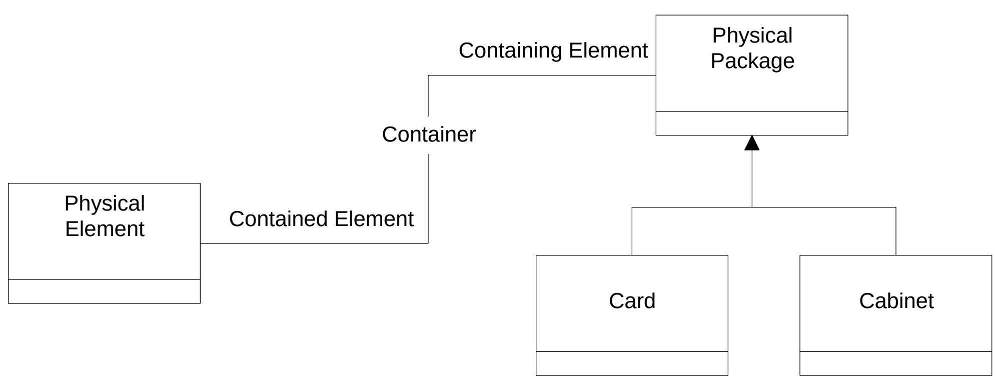


Figure 5 – References, Ranges, Domains, and Inheritance

A similar relationship can exist between properties. For example, given that PhysicalPackage has a Name property (which is a simple alphanumeric string); Card overrides Name to an alpha-only string. Similarly, a method that overrides another method must support the same signature as the original method and, most importantly, must be a substitute for the original method in all cases.

26) The override relationship is used to indicate the substitution relationship between a property or method of a subclass and the overridden property or method inherited from the superclass. This is the opposite of the C++ convention in which the superclass property or method is specified as virtual, with overrides as a side effect of declaring a feature with the same signature as the inherited virtual feature.

27) The number of references in an association class defines the arity of the association. An association containing two references is a binary association. An association containing three references is a ternary Association. Unary associations, which contain one reference, are not meaningful. Arrays of references are not allowed. When an association is subclassed, its arity cannot change.

28) Schemas allow ownership of portions of the overall model by individuals and organizations who manage the evolution of the schema. In any given installation, all classes are visible, regardless of schema ownership. Schemas have a universally unique name. The schema name is part of the class name. The full class name (that is, class name plus owning schema name) is unique within the namespace and is the fully-qualified name (see 5.4).

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) may have the special value NULL, indicating absence of value, unless further constrained in this document.

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	UCS-2 string
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 string containing a date-time
<classname> ref	Strongly typed reference
char16	16-bit UCS-2 character

5.2.1 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 a + (plus) or – (minus), indicating that the value is a timestamp with the sign of Universal Coordinated Time (UTC), which is basically the same as Greenwich Mean Time correction field. A + (plus) is used for time zones east of Greenwich, and a – (minus) is used for time zones west of Greenwich.
- utc is the offset from UTC in minutes (using the sign indicated by s).

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

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.

For example, Monday, May 25, 1998, at 1:30:15 PM EST is represented as 19980525133015.0000000-300.

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

The format for intervals is as follows:

ddddddddhhmmss.mmmmmm:000, with

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 properties.

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 that 1 BCE is the year before 1 CE.

2) The expression is evaluated using the following rules for any datetime ranges:

- Definitions:

  T(x, y) The microsecond range for a timestamp with the lower bound x and the upper bound y

  I(x, y) The microsecond range for an interval with the lower bound x and the upper bound y

  D(x, y) The microsecond range for a datetime (timestamp or interval) with the lower bound x and the upper bound y

- Rules:

  I(a, b) + I(c, d) := I(a+c, b+d)
  I(a, b) - I(c, d) := I(a-d, b-c)
  T(a, b) + I(c, d) := T(a+c, b+d)
  T(a, b) - I(c, d) := T(a-d, b-c)
  T(a, b) - T(c, d) := I(a-d, b-c)
  I(a, b) * c := I(a*c, b*c)
  I(a, b) / c := I(a/c, b/c)

  D(a, b) < D(c, d) := true if b < c, false if a >= d, otherwise NULL (uncertain)
  D(a, b) <= D(c, d) := true if b <= c, false if a > d, otherwise NULL (uncertain)
  D(a, b) > D(c, d) := true if a > d, false if b <= c, otherwise NULL (uncertain)
  D(a, b) >= D(c, d) := true if a >= d, false if b < c, otherwise NULL (uncertain)
  D(a, b) = D(c, d) := true if a = b = c = d, false if b < c OR a > d, otherwise NULL (uncertain)
  D(a, b) <> D(c, d) := true if b < c OR a > d, false if a = b = c = d, otherwise NULL (uncertain)

  These rules follow the well-known mathematical interval arithmetic. For a definition of mathematical interval arithmetic, see http://en.wikipedia.org/wiki/Interval_arithmetic.

  NOTE 1: Mathematical interval arithmetic is commutative and associative for addition and multiplication, as in ordinary arithmetic.

  NOTE 2: Mathematical interval arithmetic mandates the use of three-state logic for the result of comparison operations. A special value called "uncertain" indicates that a decision cannot be made. The special value of "uncertain" is mapped to the NULL value in datetime comparison operations.

3) Overflow and underflow condition checking 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.

EXAMPLE:

```
"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
"200510031l****.******+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"

5.2.2 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.

5.3 Supported Schema Modifications

Some of the following supported schema modifications change application behavior. Changes are all subject to security restrictions. Only the owner of the schema or someone authorized by the owner can modify the schema.

- A class can be added to or deleted from a schema.
- A property can be added to or deleted from a class.
- A class can be added as a subtype or supertype of an existing class.
- A class can become an association as a result of the addition of an Association qualifier, plus two or more references.
- A qualifier can be added to or deleted from any named element to which it applies.
- The Override qualifier can be added to or removed from a property or reference.
- A method can be added to a class.
- A method can override an inherited method.
- Methods can be deleted, and the signature of a method can be changed.
- A trigger may be added to or deleted from a class.

In defining an extension to a schema, the schema designer is expected to operate within the constraints of the classes defined in the core model. It is recommended that any added component of a system be defined as a subclass of an appropriate core model class. For each class in the core model, the schema designer is expected to consider whether the class being added is a subtype of this class. After the core model class to be extended is identified, the same question should be addressed for each subclass of the identified class. This process defines the superclasses of the class to be defined and should be continued until the most detailed class is identified. The core model is not a part of the meta schema, but it is an important device for introducing uniformity across schemas that represent aspects of the managed environment.

5.3.1 Schema Versions

Schema versioning is described in the 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 5.5.2.53 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 3 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 3 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 3 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.

Table 3 – 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 a Required qualifier	
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.
Change in the following qualifiers: In/Out, Units	

5.4 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

5.5 Qualifiers

Qualifiers are values that provide additional information about classes, associations, indications, methods, method parameters, properties, or references. Qualifiers shall not be applied to qualifiers or to qualifier types. All qualifiers have a name, type, value, scope, flavor, and default value. Qualifiers cannot be duplicated. There cannot be more than one qualifier of the same name for any given class, association, indication, method, method parameter, property, or reference.

The following clauses describe meta, standard, optional, and user-defined qualifiers. When any of these qualifiers are used in a model, they must be declared in the MOF file before they are used. These declarations must abide by the details (name, applied to, type) specified in the tables below. It is not valid to change any of this information for the meta, standard, or optional qualifiers. The default values can be changed. A default value is the assumed value for a qualifier when it is not explicitly specified for particular model elements.

5.5.1 Meta Qualifiers

Table 4 lists the qualifiers that refine the definition of the meta constructs in the model. These qualifiers refine the actual usage of a class declaration and are mutually exclusive.

Table 4 – Meta Qualifiers

Qualifier	Default	Type	Description
Association	FALSE	Boolean	The object class is defining an association.
Indication	FALSE	Boolean	The object class is defining an indication.

5.5.2 Standard Qualifiers

The following subclauses list the standard qualifiers required for all CIM-compliant implementations. Any given object does not have all the qualifiers listed. 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 5.5.1), 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 5.5.2.3) 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.

5.5.2.1 Abstract

The Abstract qualifier takes Boolean values, and has a Scope(Class Association Indication). 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.

5.5.2.2 Aggregate

The Aggregate qualifier takes Boolean values, and has a Scope(Reference). 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.

5.5.2.3 Aggregation

The Aggregation qualifier takes Boolean values, and has Scope(Association). The default value is FALSE.

The Aggregation qualifier indicates that the association is an aggregation.

5.5.2.4 ArrayType

The ArrayType qualifier takes string array values, and has Scope(Property Parameter). The default value is FALSE.

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).

5.5.2.5 Bitmap

The Bitmap qualifier takes string array values, and has a Scope(Property Parameter Method). 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.

5.5.2.6 BitValues

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

The BitValues qualifier translates between a bit position value and an associated string. See 5.5.2.5 for the description for the Bitmap qualifier.

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

5.5.2.7 ClassConstraint

The ClassConstraint qualifier takes string array values and has Scope(Class Association Indication). The default value is NULL.

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

    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 syntax:

    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: 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())"
}
```

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)"
}
```

5.5.2.8 Composition

The Composition qualifier takes Boolean values and has Scope(Association). 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 *OMG UML Specification*. Following is a quote (with emphasis added) from 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.

5.5.2.9 Correlatable

The Correlatable qualifier takes string array values, and has Scope(Property). 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:

```
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 that 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.

5.5.2.10 Counter

The Counter qualifier takes string array values and has Scope(Property Parameter Method). 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.

5.5.2.11 Deprecated

The Deprecated qualifier takes string array values and has Scope(Class Association Indication Property Reference Parameter Method). 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 instrumentation shall continue to support the deprecated element so that current applications do not break. Existing instrumentation should add support for any replacement elements. A deprecated element should not be used in new applications. Existing and new applications 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 syntax:

```
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 management application 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.

5.5.2.12 Description

The Description qualifier takes string array values, and has a Scope(Class Association Indication Property Reference Parameter Method). The default value is NULL.

The Description qualifier describes a named element.

5.5.2.13 DisplayName

The DisplayName qualifier takes string values and has Scope(Class Association Indication Property Reference Parameter Method). 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.

5.5.2.14 DN

The DN qualifier takes string array values, and has a Scope(Property Parameter Method). 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 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.

5.5.2.15 EmbeddedInstance

The EmbeddedInstance qualifier takes string array values and has Scope(Property Parameter Method). The default value is NULL.

The qualified string typed element contains an embedded instance. The encoding of the instance contained in the string typed element qualified by EmbeddedInstance follows the rules defined in ANNEX G.

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

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.

This qualifier shall not be used on an element that overrides an element not qualified by EmbeddedInstance. However, it may be used on an overriding element to narrow the class specified in this qualifier on the overridden element to one of its subclasses.

See ANNEX G for examples.

5.5.2.16 EmbeddedObject

The EmbeddedObject qualifier takes Boolean values and has Scope(Property Parameter Method). 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 follows the rules defined in ANNEX G.

This qualifier may be used only on elements of string type. It shall not be used on an element that overrides an element not qualified by EmbeddedObject.

See ANNEX G for examples.

5.5.2.17 Exception

The Exception qualifier takes Boolean values and has Scope(Class Indication). 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 CIMEntities. The Exception qualifier cannot be used with the Abstract qualifier. The subclass of an exception class shall be an exception class.

5.5.2.18 Experimental

The Experimental qualifier takes Boolean values and has Scope(Class Association Indication Property Reference Parameter Method). 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.

5.5.2.19 Gauge

The Gauge qualifier takes Boolean values and has Scope(Property Parameter Method). 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.

5.5.2.20 IN

The IN qualifier takes Boolean values and has Scope(Parameter). The default value is TRUE.

The IN qualifier is used with an associated parameter to pass values to a method.

5.5.2.21 IsPUnit

The IsPUnit qualifier takes Boolean values and has Scope(Property Parameter Method). 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.

Experimental: This qualifier has status "Experimental."

5.5.2.22 Key

The Key qualifier takes Boolean values and has Scope(Property Reference). The default value is FALSE.

The property or reference is part of the model path (see 8.3.2 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.

5.5.2.23 MappingStrings

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

This qualifier indicates mapping strings for one or more management data providers or agents. See 5.5.5 for details.

5.5.2.24 Max

The Max qualifier takes uint32 values and has Scope(Reference). 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.

5.5.2.25 MaxLen

The MaxLen qualifier takes uint32 values and has Scope(Property Parameter Method). 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.

5.5.2.26 MaxValue

The MaxValue qualifier takes uint32 values and has Scope(Property Parameter Method). 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.

5.5.2.27 MethodConstraint

The MethodConstraint qualifier takes string array values and has Scope(Method). The default value is NULL.

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

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 syntax:

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 syntax:

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 EnabledLogicalElement class specifies that if a Job parameter is returned as not NULL, then an OwningJobElement association must exist between the EnabledLogicalElement class and the Job.

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

5.5.2.28 Min

The Min qualifier takes uint32 values and has Scope(Reference). 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.

5.5.2.29 MinLen

The MinLen qualifier takes uint32 values and has Scope(Property Parameter Method). 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.

5.5.2.30 MinValue

The MinValue qualifier takes sint64 values and has Scope(Property Parameter Method). 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.

5.5.2.31 ModelCorrespondence

The ModelCorrespondence qualifier takes string array values and has Scope(Class Association Indication Property Reference Parameter Method). 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 syntax:

<className> [*("."(<propertyName> | < referenceName>)) ["." <methodName> ["(" <parameterName> ")"]]]

Note that 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 may only be on the vendor's Indication class, which is an extension to CIM.
- 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.

5.5.2.32 NonLocal

This instance-level qualifier and the corresponding pragma were removed as an erratum by CR1461.

5.5.2.33 NonLocalType

This instance-level qualifier and the corresponding pragma were removed as an erratum by CR1461.

5.5.2.34 NullValue

The NullValue qualifier takes string values and has Scope(Property). 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 integer value. The syntax for representing an integer value is:

    ["+" / "-"] 1*<decimalDigit>

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

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

5.5.2.35 OctetString

The OctetString qualifier takes Boolean values and has Scope(Property Parameter Method). The default value is FALSE.

This qualifier identifies the qualified property or parameter as an octet string.

When used in conjunction with an unsigned 8-bit integer (uint8) array, the OctetString qualifier indicates that the unsigned 8-bit integer array represents a single octet string.

When used in conjunction with arrays of strings, the OctetString qualifier indicates that the qualified character strings are encoded textual conventions representing octet strings. The text encoding of these binary values conforms to the following grammar: "0x" 4*(<hexDigit> <hexDigit>). In both cases, the first 4 octets of the octet string (8 hexadecimal digits in the text encoding) are the number of octets in the represented octet string with the length portion included in the octet count. (For example, "0x00000004" is the encoding of a 0 length octet string. A second example is "0x000000050A" that is an encoding of the octect string “0x0A”.)

5.5.2.36 Out

The Out qualifier takes Boolean values and has Scope(Parameter). The default value is FALSE.

The Out qualifier indicates that the associated parameter is used to return values from a method.

5.5.2.37 Override

The Override qualifier takes string values and has Scope(Property Parameter Method). 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 have the following format:

[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 shall be present if the class exposes more than one element with the same name. (See 7.5.1.)

If the 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 the 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.

5.5.2.38 Propagated

The Propagated qualifier takes string values and has Scope(Property). The default value is NULL.

The Propagated qualifier is a string-valued qualifier that contains the name of the key that is propagated. Its use assumes only one Weak qualifier on a reference with the containing class as its target. The associated property shall have the same value as the property named by the qualifier in the class on the other side of the weak association. The format of the string to accomplish this is as follows:

[<className> "."] <IDENTIFIER>

When the Propagated qualifier is used, the Key qualifier shall be specified with a value of TRUE.

5.5.2.39 PropertyConstraint

The PropertyConstraint qualifier takes string array values and has Scope(Property Reference). 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 OMG *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 syntax:

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 syntax:

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 PolicySetInSystem. The following qualifier defined on PolicyAction.SystemName specifies that constraint:

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

A property shall not be qualified with more than one initialization constraint or derivation constraint. The definition of an initialization constraint and a derivation constraint on the same property is allowed. In this case, the value of the property immediately after creation of the instance shall satisfy both constraints.

5.5.2.40 PUnit

The PUnit qualifier takes string array values and has Scope(Property Parameter Method). The default value is NULL.

The PUnit qualifier indicates the programmatic unit of measure of the qualified property, method return value, or method parameter. The qualifier value follows the syntax for programmatic units.

NULL indicates that the programmatic unit is unknown. The syntax for programmatic units is defined in ANNEX C.

Experimental: This qualifier has a status of "Experimental."

5.5.2.41 Read

The Read qualifier takes Boolean values and has Scope(Property). The default value is TRUE.

The Read qualifier indicates that the property is readable.

5.5.2.42 Required

The Required qualifier takes Boolean values and has Scope(Property Reference Parameter Method). 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 applications 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).

5.5.2.43 Revision (Deprecated)

The Revision qualifier is deprecated. (See 5.5.2.53 for the description of the Version qualifier.)

The Revision qualifier takes string values and has Scope(Class Association Indication). 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.

5.5.2.44 Schema (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 and has Scope(Property Method). The default value is NULL.

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

5.5.2.45 Source

This instance-level qualifier and the corresponding pragma are removed as an erratum by CR1461.

5.5.2.46 SourceType

This instance-level qualifier and the corresponding pragma are removed as an erratum by CR1461.

5.5.2.47 Static

The Static qualifier takes Boolean values and has Scope(Property Method). The default value is FALSE.

The property or method is static. For a definition of static properties, see 7.5.6. For a definition of static methods, see 7.9.1.

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

5.5.2.48 Terminal

The Terminal qualifier takes Boolean values and has Scope(Class Association Indication). 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.

5.5.2.49 UMLPackagePath

The UMLPackagePath qualifier takes string values and has Scope(Class Association Indication). 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".

5.5.2.50 Units (Deprecated)

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

The Units qualifier takes string values and has Scope(Property Parameter Method). 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.

5.5.2.51 ValueMap

The ValueMap qualifier takes string array values and has Scope(Property Parameter Method). 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.

Where:

ValueMap may be used only with string or integer types.

When used with a string type, a ValueMap entry is a MOF stringvalue.

When used with an integer type, a ValueMap entry is a MOF integervalue or an integervaluerange as defined here.

```
integervaluerange:
    [integervalue] ".." [integervalue]
```

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 type of the associated property.

ValueMap = ("..") is not permitted.

If used with a Value array, then all values claimed by a particular ValueMap entry apply to the corresponding Value 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. (Note, however, that 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.

5.5.2.52 Values

The Values qualifier takes string array values and has Scope(Property Parameter Method). 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.

5.5.2.53 Version

The Version qualifier takes string values and has Scope(Class Association Indication). 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, or, more formally, 1*<decimalDigit> "." 1*<decimalDigit> "." 1*<decimalDigit>

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.

EXAMPLE:

```
Version("2.7.0")
Version("1.0.0")
```

5.5.2.54 Weak

The Weak qualifier takes Boolean values and has Scope(Reference). The default value is FALSE.

The keys of the referenced class include the keys of the other participants in the association. This qualifier is used when the identity of the referenced class depends on that of the other participants in the association. No more than one reference to any given class can be weak. The other classes in the association shall define a key. The keys of the other classes are repeated in the referenced class and tagged with a propagated qualifier.

5.5.2.55 Write

The Write qualifier takes Boolean values and has Scope(Property). 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.

5.5.3 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.

5.5.3.1 Alias

The Alias qualifier takes string values and has Scope(Property Reference Method). The default value is NULL.

The Alias qualifier establishes an alternate name for a property or method in the schema.

5.5.3.2 Delete

The Delete qualifier takes Boolean values and has Scope(Association Reference). 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.

Applications 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.

5.5.3.3 DisplayDescription

The DisplayDescription qualifier takes string values and has Scope(Class Association Indication Property Reference Parameter Method). 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.

5.5.3.4 Expensive

The Expensive qualifier takes string values and has Scope(Class Association Indication Property Reference Parameter Method).The default value is FALSE.

The Expensive qualifier indicates that the element is expensive to manipulate and/or compute.

5.5.3.5 IfDeleted

The IfDeleted qualifier takes Boolean values and has Scope(Association Reference). 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.

5.5.3.6 Invisible

The Invisible qualifier takes Boolean values and has Scope(Class Association Property Reference Method). 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.

5.5.3.7 Large

The Large qualifier takes Boolean values and has Scope(Class Property). The default value is FALSE.

The Large qualifier property or class requires a large amount of storage space.

5.5.3.8 PropertyUsage

The PropertyUsage qualifier takes string values and has Scope(Property). 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”.

5.5.3.9 Provider

The Provider qualifier takes string values and has Scope(Class Association Indication Property Reference Parameter Method). The default value is NULL.

An implementation-specific handle to the instrumentation that populates elements in the schemas that refers to dynamic data.

5.5.3.10 Syntax

The Syntax qualifier takes string values and has Scope(Property, Reference, Parameter Method). 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.

5.5.3.11 SyntaxType

The SyntaxType qualifier takes string values and has Scope(Property Reference Parameter Method). The default value is NULL.

The SyntaxType qualifier defines the format of the Syntax qualifier. It must be used with the Syntax qualifier.

5.5.3.12 TriggerType

The TriggerType qualifier takes string values and has Scope(Class Association Indication Property Reference Method). 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.

5.5.3.13 UnknownValues

The UnknownValues qualifier takes string values and has Scope(Property). 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.

5.5.3.14 UnsupportedValues

The UnsupportedValues qualifier takes string values and has Scope(Property). 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.

5.5.4 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.

5.5.5 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 specification. Extension schemas should use only the mapping string formats defined in this specification.

The mapping string formats defined in this specification conform to the following formal syntax:

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.

5.5.5.1 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:

```
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 tokens in mib_format should be assembled without intervening white space characters. 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.

For 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:

```
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").

The tokens in oid_format should be assembled without intervening white space characters.

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

5.5.5.2 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. Note that 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

general_format_fullname = general_format_name "." general_format_defining_body
```

Where:

```
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 tokens in general_format and general_format_fullname should be assembled without intervening white space characters.

The text in Figure 6 is an example that defines a mapping 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:
>
> ```
> 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.
>
> The tokens in general_format_mapping and general_format_fullname should be assembled without intervening white space characters.

Figure 6 – Example for Mapping a String Format Based on the General Mapping String Format

5.5.5.3 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.

Deprecation Note: 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 specification is also considered deprecated. The MappingStrings qualifier itself is not deprecated because it is used for formats other than MIF.

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:

```
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.

By default, the formal syntax rules in this (current) specification allow each token to be separated by an arbitrary number of white spaces. However, 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 white space 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:

```
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:

```
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.

NOTE: All grammars defined in this specification use the notation defined in RFC 4234; any exceptions are stated with the grammar.

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.

A MOF file can be encoded in either Unicode or UTF-8.

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 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 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 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 can 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. 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 names, see ANNEX E.

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 in a class definition can have default initializers. For example:

```
class Acme_Disk_v1 : CIM_Media
{
     string Manufacturer = "Acme";
     string ModelNumber  = "123-AAL";
};
```

When new instances of the class are declared, any such property is automatically assigned its default value unless the instance declaration explicitly assigns a value to the property.

7.5.4 Class and Property Qualifiers

Qualifiers are meta data about a property, method, method parameter, or class, and they are not part of the definition itself. For example, a qualifier indicates whether a property value can be changed (using the Write qualifier). Qualifiers always precede the declaration to which they apply.

Certain qualifiers are well known and cannot be redefined (see 5.5). Apart from these restrictions, arbitrary qualifiers may be used.

Qualifier declarations include an explicit type indicator, which must be one of the intrinsic types. A qualifier with an array-based parameter is assumed to have a type, which is a variable-length homogeneous array of one of the intrinsic types. In Boolean arrays, each element in the array is either TRUE or FALSE.

EXAMPLE:

```
Write(true)                          // boolean
profile { true, false, true }        // boolean []
description("A string")              // string
info { "this", "a", "bag", "is" }    // string []
id(12)                               // uint32
idlist { 21, 22, 40, 43 }            // uint32 []
apple(3.14)                          // real32
oranges { -1.23E+02, 2.1 }           // real32 []
```

Qualifiers are applied to a class by preceding the class declaration with a qualifier list, comma-separated and enclosed within square brackets. Qualifiers are applied to a property or method in a similar way.

EXAMPLE:

```
class CIM_Process:CIM_LogicalElement
{
     uint32 Priority;
       [Write(true)]
     string Handle;
};
```

When a Boolean qualifier is specified in a class or property declaration, the name of the qualifier can be used without also specifying a value. From the previous example:

```
class CIM_Process:CIM_LogicalElement
{
   uint32 Priority;
      [Write] // Equivalent declaration to Write (True)
   string Handle;
};
```

If only the qualifier name is listed for a Boolean qualifier, it is implicitly set to TRUE. In contrast, when a qualifier is not specified at all for a class or property, the default value for the qualifier is assumed. Consider another example:

```
    [Association,
    Aggregation]    // Specifies the Aggregation qualifier to be True
class CIM_SystemDevice: CIM_SystemComponent
{
        [Override ("GroupComponent"),
        Aggregate]  // Specifies the Aggregate qualifier to be True
    CIM_ComputerSystem Ref GroupComponent;
        [Override ("PartComponent"),
        Weak] // Defines the Weak qualifier to be True
    CIM_LogicalDevice Ref PartComponent;
};

[Association]    // Since the Aggregation qualifier is not specified,
                 // its default value, False, is set
class Acme_Dependency: CIM_Dependency
{
        [Override ("Antecedent")]    // Since the Aggregate and Weak
                                     // qualifiers are not used, their
                                     // default values, False, are assumed
```

```
        Acme_SpecialSoftware Ref Antecedent;
            [Override ("Dependent")]
        Acme_Device Ref Dependent;
    };
```

Qualifiers can automatically be transmitted from classes to derived classes or from classes to instances, subject to certain rules. The rules prescribing how the transmission occurs are attached to each qualifier and encapsulated in the concept of the qualifier flavor. For example, a qualifier can be designated in the base class as automatically transmitted to all of its derived classes, or it can be designated as belonging specifically to that class and not transmittable. The former is achieved by using the ToSubclass flavor, and the latter by using the Restricted flavor. These two flavors shall not be used at the same time. In addition, if a qualifier is transmitted to its derived classes, the qualifier flavor can be used to control whether derived classes can override the qualifier value or whether the qualifier value must be fixed for an entire class hierarchy. This aspect of qualifier flavor is referred to as override permissions.

Override permissions are assigned using the EnableOverride or DisableOverride flavors, which shall not be used at the same time. If a qualifier is not transmitted to its derived classes, these two flavors are meaningless and shall be ignored.

Qualifier flavors are indicated by an optional clause after the qualifier and are preceded by a colon. They consist of some combination of the key words EnableOverride, DisableOverride, ToSubclass, and Restricted, indicating the applicable propagation and override rules.

EXAMPLE:

```
    class CIM_Process:CIM_LogicalElement
    {
        uint32 Priority;
            [Write(true):DisableOverride ToSubclass]
        string Handle;
    };
```

In this example, Handle is designated as writable for the Process class and for every subclass of this class.

The recognized flavor types are shown in Table 5.

Table 5 – Recognized Flavor Types

Parameter	Interpretation	Default
ToSubclass	The qualifier is inherited by any subclass.	ToSubclass
Restricted	The qualifier applies only to the class in which it is declared.	ToSubclass
EnableOverride	If ToSubclass is in effect, the qualifier can be overridden.	EnableOverride
DisableOverride	If ToSubclass is in effect, the qualifier cannot be overridden.	EnableOverride
Translatable	The value of the qualifier can be specified in multiple locales (language and country combination). When Translatable(yes) is specified for a qualifier, it is legal to create implicit qualifiers of the form: label_ll_cc where ▪ label is the name of the qualifier with Translatable(yes). ▪ ll is the language code for the translated string. ▪ cc is the country code for the translated string. In other words, a label_ll_cc qualifier is a clone, or derivative, of the "label" qualifier with a postfix to capture the locale of the translated value. The locale of the original value (that is, the value specified using the qualifier with a name of "label") is determined by the locale pragma. When a label_ll_cc qualifier is implicitly defined, the values for the other flavor parameters are assumed to be the same as for the "label" qualifier. When a label_ll_cc qualifier is explicitly defined, the values for the other flavor parameters must also be the same. A "yes" for this parameter is valid only for string-type qualifiers. EXAMPLE: If an English description is translated into Mexican Spanish, the actual name of the qualifier is: DESCRIPTION_es_MX.	No

7.5.5 Key Properties

Instances of a class require a way to distinguish the instances within a single namespace. Designating one or more properties with the reserved Key qualifier provides 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;
};
```

In this example, instances of Drive are distinguished using the Volume property, which acts as the key for the class.

Compound keys are supported and are designated by marking each of the required properties with the key qualifier.

If a new subclass is defined from a superclass and the superclass has key properties (including those inherited from other classes), the new subclass *cannot* define any additional key properties. New key properties in the subclass can be introduced only if all classes in the inheritance chain of the new subclass are keyless.

If any reference to the class has the Weak qualifier, the properties that are qualified as Key in the other classes in the association are propagated to the referenced class. The key properties are duplicated in the referenced class using the name of the property, prefixed by the name of the original declaring class. For example:

```
class CIM_System:CIM_LogicalElement
{
        [Key]
    string Name;
};

class CIM_LogicalDevice: CIM_LogicalElement
{
     [Key]
    string DeviceID;
        [Key, Propagated("CIM_System.Name")]
    string SystemName;
};

[Association]
class CIM_SystemDevice: CIM_SystemComponent
{
        [Override ("GroupComponent"), Aggregate, Min(1), Max(1)]
    CIM_System Ref GroupComponent;
        [Override ("PartComponent"), Weak]
    CIM_LogicalDevice Ref PartComponent;
};
```

7.5.6 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 5.5.2). 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 7.6.2.)

7.6.3 Key References and Properties

Instances of an association also must provide a way to distinguish the instances, for they are just a special kind of a class. Designating one or more references/properties with the reserved Key qualifier identifies the instances.

A reference/property of an association is (part of) the association key if the Key qualifier is applied.

```
    [Association, Aggregation]
class CIM_Component
{
        [Aggregate, Key]
    CIM_ManagedSystemElement Ref GroupComponent;
        [Key]
    CIM_ManagedSystemElement Ref PartComponent;
};
```

The key definition of association follows the same rules as for ordinary classes. Compound keys are supported in the same way. Also a new subassociation *cannot* define additional key properties/references. If any reference to a class has the Weak qualifier, the KEY-qualified properties of the other class, whose reference is not Weak-qualified, are propagated to the class (see 7.5.5).

7.6.4 Object References

Object references are special properties whose values are links or pointers to other objects (classes or instances). The value of an object reference is expressed as a string, which represents a path to another object. A non-NULL value of an object reference includes:

- The namespace in which the object resides
- The class name of the object
- The values of all key properties for an instance if the object represents an instance

The data type of an object reference is declared as "XXX ref", indicating a strongly typed reference to objects of the class with name "XXX" or a derivation of this class. For example:

```
    [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.

References in associations shall not have the special NULL value.

Also, see 7.12.2 for information about initializing references using aliases.

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 6.

Table 6 – 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 Qualifier Declarations

Qualifiers may be declared using the keyword "qualifier." The declaration of a qualifier allows the definition of types, default values, propagation rules (also known as Flavors), and restrictions on use.

The default value for a declared qualifier is used when the qualifier is not explicitly specified for a given schema element. Explicit specification includes inherited qualifier specification.

The MOF syntax allows a qualifier to be specified without an explicit value. The assumed value depends on the qualifier type: Boolean types are TRUE, numeric types are NULL, strings are NULL, and arrays are empty. For example, the Alias qualifier is declared as follows:

```
qualifier alias :string = null, scope(property, reference, method);
```

This declaration establishes a qualifier called alias of type string. It has a default value of NULL and may be used only with properties, references, and methods.

The meta qualifiers are declared as follows:

```
Qualifier Association : boolean = false,
    Scope(class, association), Flavor(DisableOverride);

Qualifier Indication : boolean = false,
    Scope(class, indication), Flavor(DisableOverride);
```

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. 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. Any property values not initialized have default values as specified in the class definition, or (if no default value is specified) the special value NULL to indicate absence of value. 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.

For subclasses, all properties in the superclass must have their values initialized along with the properties in the subclass. Any property values not specifically assigned in the instance block have either the default value for the property (if there is one) or the value NULL.

The values of all key properties must be specified for an instance to be identified and created. There is no requirement to initialize other property values explicitly. See 7.11.6 for information on behavior when there is no property value initialization.

As described in item 21)-e) of 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 is scoped 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 I for more information about the name conflict issue.

Instances of associations may also be defined, as in the following example:

```
instance of CIM_ServiceSAPDependency
{
   Dependent = "CIM_Service.Name = \"mail\"";
   Antecedent = "CIM_ServiceAccessPoint.Name = \"PostOffice\"";
};
```

7.8.1 Instance Aliasing

An alias can be assigned to an instance using this syntax:

```
instance of Acme_LogicalDisk as $Disk
{
   // Body of instance definition here ...
};
```

Such an alias can later be used within the same MOF specification as a value for an object reference property. For more information, see 7.12.2*.*

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 side for output parameters. The array is thought to be retrieved and deleted by the CIM server side 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.

Like any CIM type, an array itself may have the special NULL value to indicate absence of value. Conceptually, the value of the array itself, if not absent, is the set of its elements. An empty array (that is, an array with no elements) must be distinguishable from an array that has the special NULL value. For example, if an array contains error messages, it makes a difference to know that there are no error messages rather than to be uncertain about whether there are any error messages.

The type of an array is defined by the ArrayType 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 side maintains the order of elements in the array as long as no array elements are added, deleted, or changed. Therefore, the CIM server side 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 side itself determines the order of elements on these occasions and therefore possibly reorders the elements. The CIM server side 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 server side again determines a new order and from then on maintains that new order. For output array parameters, the server side determines the order of elements and the client side 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 side. The order the CIM server side 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 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 syntax is used:

```
class 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};
};
```

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 specification.

In the following example, Start and Stop methods are defined on the Service class. Each method returns an integer value:

```
class CIM_Service:CIM_LogicalElement
{
        [Key]
    string Name;
    string StartMode;
    boolean Started;
    uint32 StartService();
    uint32 StopService();
};
```

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 7.

Table 7 – 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 the language code based on ISO/IEC 639 and cc is the country code based on ISO/IEC 3166.
#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 the language code based on ISO/IEC 639, and cc is the country code based on ISO/IEC 3166. When the pragma is not specified, the assumed locale is "en_US". 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 are removed as errata by CR1461.
#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 is a sequence of zero or more UCS-2 characters enclosed in double-quotes ("). A double-quote is allowed within the value, as long as it is preceded immediately by a backslash (\).

For example, the following is a string constant:

```
"This is a string"
```

Successive quoted strings are concatenated as long as only white space or a comment intervenes:

```
"This"  " becomes a long string"
"This" /* comment */ " becomes a long string"
```

Escape sequences are recognized as legal characters within a string. The complete set of escape sequences is as follows:

```
\b          // \x0008: backspace BS
\t          // \x0009: horizontal tab HT
\n          // \x000A: linefeed LF
\f          // \x000C: form feed FF
\r          // \x000D: carriage return CR
\"          // \x0022: double quote "
\'          // \x0027: single quote '
\\          // \x005C: backslash \
\x<hex>     // where <hex> is one to four hex digits
\X<hex>     // where <hex> is one to four hex digits
```

The character set of the string depends on the character set supported by the local installation. While the MOF specification may be submitted in UCS-2 form defined in ISO/IEC 10646:2003, the local implementation may only support ANSI and *vice versa*. Therefore, the string type is unspecified and dependent on the character set of the MOF specification itself. If a MOF specification is submitted using UCS-2 characters outside the normal ASCII range, the implementation may have to convert these characters to the locally-equivalent character set.

7.11.2 Character Constants

Character and wide-character constants are specified as follows:

```
'a'
'\n'
'1'
'\x32'
```

Forms such as octal escape sequences (for example, '\020') are not supported. Integer values can also be used as character constants, as long as they are within the numeric range of the character type. For example, wide-character constants must fall within the range of 0 to 0xFFFF.

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
```

Note that 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

Object references are simple URL-style links to other objects, which may be classes or instances. They take the form of a quoted string containing an object path that is a combination of a namespace path and the model path. For example:

```
"//./root/default:LogicalDisk.SystemName=\"acme\",LogicalDisk.Drive=\"C\""
"//./root/default:NetworkCard=2"
```

An object reference can also be an alias. See 7.12.2 for details.

7.11.6 NULL

All types can be initialized to the predefined constant NULL, which indicates that no value is provided. The details of the internal implementation of the NULL value are not mandated by this document.

7.12 Initializers

Initializers are used in both class declarations for default values and instance declarations to initialize a property to a value. The format of initializer values is specified in clause 5 and its subclauses. The initializer value shall match the property data type. The only exceptions are the NULL value, which may be used for any data type, and integral values, which are used for characters.

7.12.1 Initializing Arrays

Arrays can be defined to be of type Bag, Ordered, or Indexed, and they can be initialized by specifying their values in a comma-separated list (as in the C programming language). The list of array elements is delimited with curly brackets. For example, given this class definition:

```
class Acme_ExampleClass
{
   [ArrayType ("Indexed")]
   string ip_addresses [];    // Indexed array of variable length
   sint32 sint32_values [10]; // Bag array of fixed length = 10
};
```

the following is a valid instance declaration:

```
instance of Acme_ExampleClass
{
   ip_addresses = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };

      // ip_address is an indexed array of at least 3 elements, where
      // values have been assigned to the first three elements of the
      // array

   sint32_values = { 1, 2, 3, 5, 6 };
};
```

Refer to 7.8.2 for additional information on declaring arrays and the distinctions between bags, ordered arrays, and indexed arrays.

7.12.2 Initializing References Using Aliases

Aliases are symbolic references to an object located elsewhere in the MOF specification. They have significance only within the MOF specification in which they are defined, and they are used only at compile time to establish references. They are not available outside the MOF specification.

An instance may be assigned an alias as described in 7.8.1. Aliases are identifiers that begin with the $ symbol. When a subsequent reference to the instance is required for an object reference property, the identifier is used in place of an explicit initializer.

Assuming that $Alias1 and $Alias2 are declared as aliases for instances and the obref1 and obref2 properties are object references, this example shows how the object references could be assigned to point to the aliased instances:

```
instance of Acme_AnAssociation
{
   strVal = "ABC";
   obref1 = $Alias1;
   obref2 = $Alias2;
};
```

Forward-referencing and circular aliases are permitted.

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 enterprise-wide identification of objects, as well as sharing of management information. CIM naming addresses the following requirements:

- Ability to locate and uniquely identify any object in an enterprise. Object names must be identifiable regardless of the instrumentation technology.
- Unambiguous enumeration of all objects.
- Ability to determine when two object names reference the same entity. This entails location transparency so that there is no need to understand which management platforms proxy the instrumentation of other platforms.
- Allow sharing of objects and instance data among management platforms. This requirement includes the creation of different scoping hierarchies that vary by time (for example, a current versus proposed scoping hierarchy).
- Facilitate move operations between object trees (including within a single management platform). Hide underlying management technology/provide technology transparency for the domain-mapping environment.

Allowing different names for DMI versus SNMP objects requires the management platform to understand how the underlying objects are implemented.

The Key qualifier is the CIM Meta-Model mechanism to identify the properties that uniquely identify an instance of a class (and indirectly an instance of an association). CIM naming enhances this base capability by introducing the Weak and Propagated qualifiers to express situations in which the keys of one object are to be propagated to another object.

8.1 Background

CIM MOF files can contain definitions of instances, classes, or both, as illustrated in Figure 7.

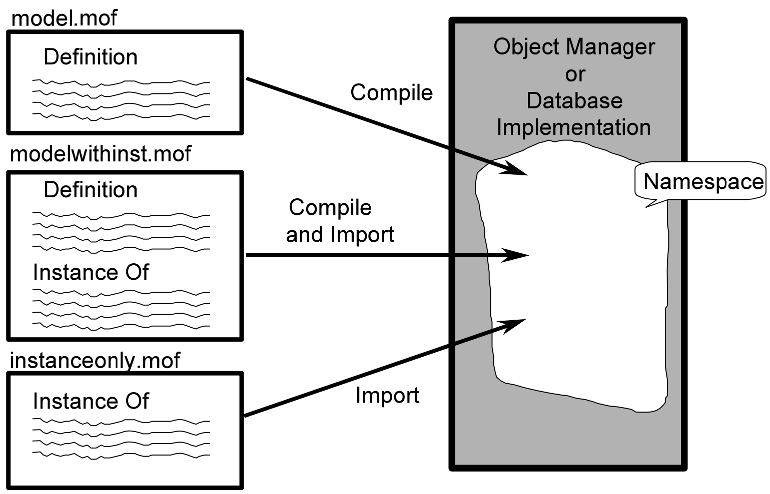


Figure 7 – Definitions of Instances and Classes

MOF files can be used to populate a technology that understands the semantics and structure of CIM. When an implementation consumes a MOF, two operations are actually performed, depending on the file's content. First, a compile or definition operation establishes the structure of the model. Second, an import operation inserts instances into the platform or tool.

When the compile and import are complete, the actual instances are manipulated using the native capabilities of the platform or tool. To manipulate an object (for example, change the value of a property), one must know the type of platform into which the information was imported, the APIs or operations used to access the imported information, and the name of the platform instance actually imported. For example, the semantics become:

> Set the Version property of the Logical Element object with Name="Cool" in the relational database named LastWeeksData to "1.4.0".

The contents of a MOF file are loaded into a namespace that provides a domain in which the instances of the classes are guaranteed to be unique per the Key qualifier definitions. The term "namespace" refers to an implementation that provides such a domain.

Namespaces can be used to accomplish the following tasks:

- Define chunks of management information (objects and associations) to limit implementation resource requirements, such as database size
- Define views on the model for applications managing only specific objects, such as hubs
- Pre-structure groups of objects for optimized query speed

Another viable operation is exporting from a particular management platform. This operation creates a MOF file for all or some portion of the information content of a platform (see Figure 8).

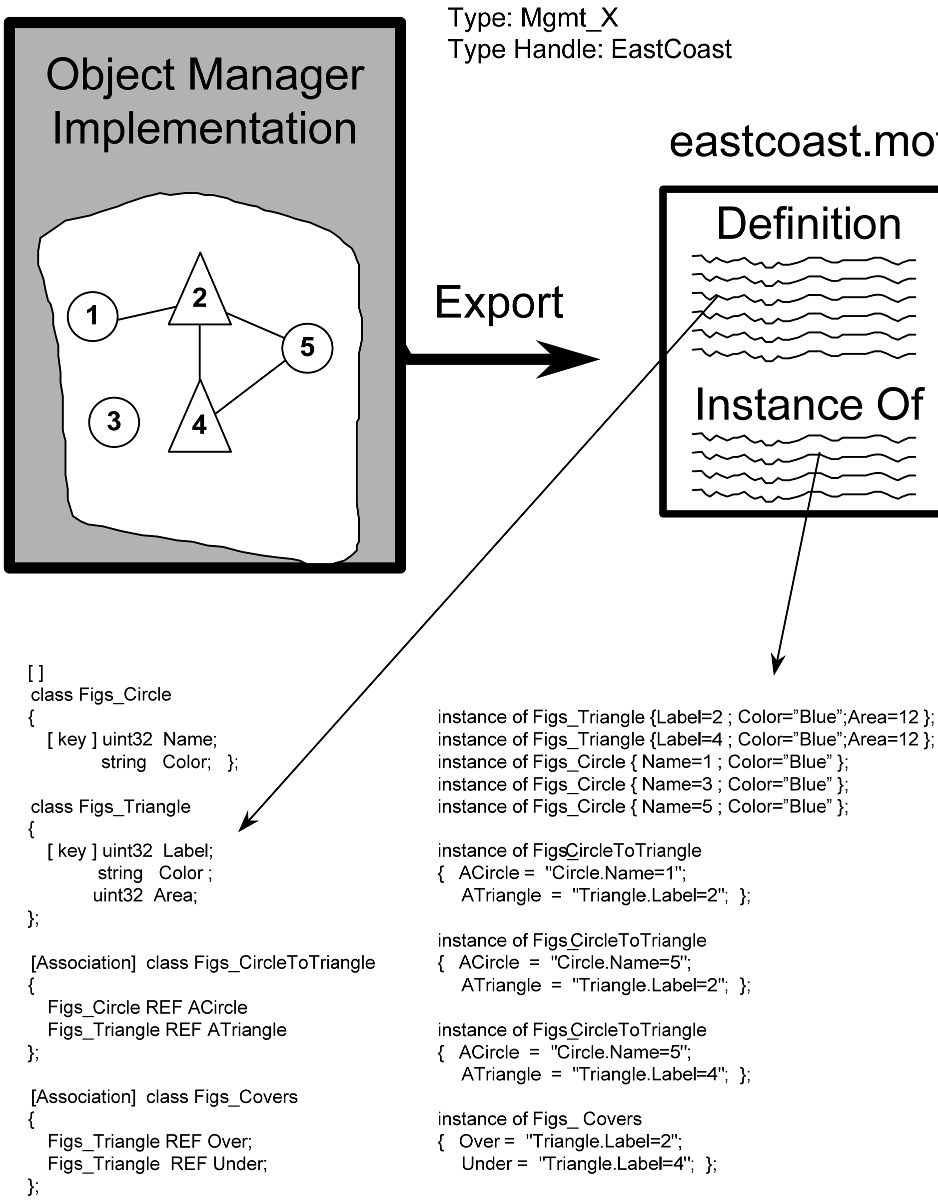


Figure 8 – Exporting to MOF

See Figure 9for an example. In this example, information is exchanged when the source system is of type Mgmt_X and its name is EastCoast. The export produces a MOF file with the circle and triangle definitions and instances 1, 3, 5 of the circle class and instances 2, 4 of the triangle class. This MOF file is then compiled and imported into the management platform of type Mgmt_ABC with the name AllCoasts.

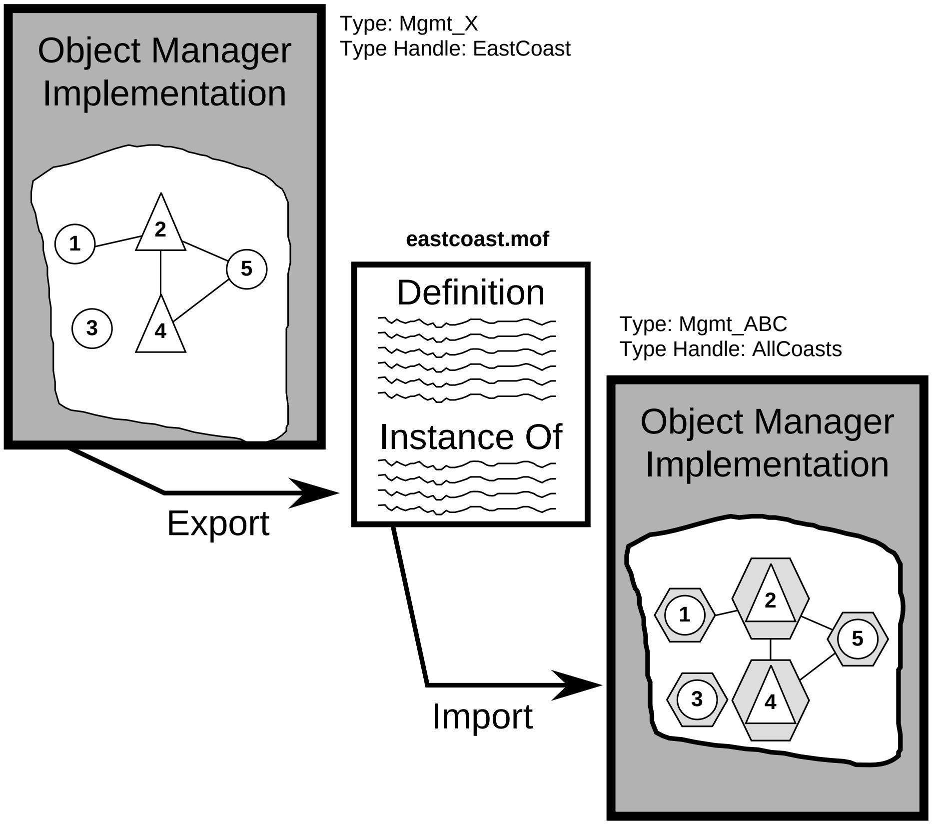


Figure 9 – Information Exchange

The import operation stores the information in a local or native format of Mgmt_ABC, so its native operations can be used to manipulate the instances. The transformation to a native format is shown in the figure by wrapping the five instances in hexagons. The transformation process must maintain the original keys.

8.1.1 Management Tool Responsibility for an Export Operation

The management tool must be able to create unique key values for each distinct object it places into the MOF file. For each instance placed into the MOF file, the management tool must maintain a mapping from the MOF file keys to the native key mechanism.

8.1.2 Management Tool Responsibility for an Import Operation

The management tool must be able to map the unique keys found in the MOF file to a set of locally-understood keys.

8.2 Weak Associations: Supporting Key Propagation

CIM provides a mechanism to name instances within the context of other object instances. For example, if a management tool handles a local system, it can refer to the C drive or the D drive. However, if a management tool handles multiple machines, it must refer to the C drive on machine X and the C drive on machine Y. In other words, the name of the drive must include the name of the hosting machine. CIM supports the notion of weak associations to specify this type of key propagation. A weak association is defined using a qualifier.

EXAMPLE:

```
Qualifier Weak: boolean = false, Scope(reference), Flavor(DisableOverride);
```

The keys of the referenced class include the keys of the other participants in the Weak association. This situation occurs when the referenced class identity depends on the identity of other participants in the association. This qualifier can be specified on only one of the references defined for an association. The weak referenced object is the one that depends on the other object for identity.

Figure 10 shows an example of a weak association. There are three classes: ComputerSystem, OperatingSystem and Local User. The Operating System class is weak with respect to the Computer System class because the runs association is marked weak. Similarly, the Local User class is weak with respect to the Operating System class, because the association is marked as weak.

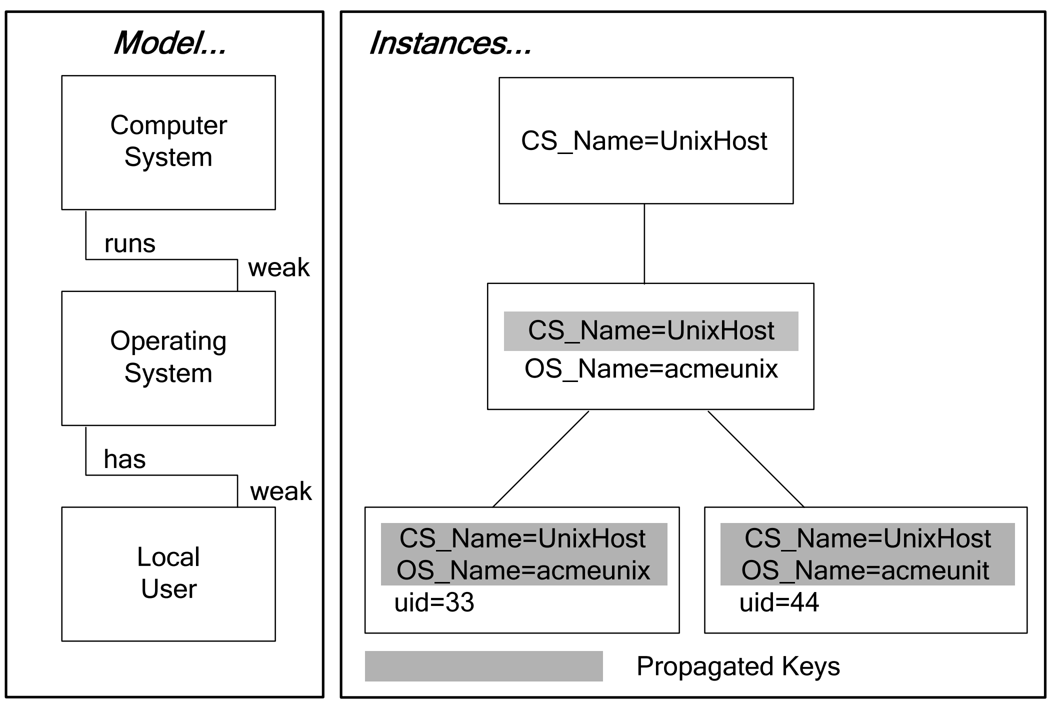


Figure 10 – Example of Weak Association

In a weak association definition, the Computer System class is a scoping class for the Operating System class because its keys are propagated to the Operating System class. The Computer System and the Operating System classes are both scoping classes for the Local User class because the Local User class gets keys from both. Finally, the Computer System is referred to as a top-level object (TLO) because it is not weak with respect to any other class. That a class is a top-level object is implied because no references to that class are marked with the Weak qualifier. In addition, TLOs must have the possibility of an enterprise-wide, unique key. For example, consider a computer's IP address in a company's enterprise-wide IP network. The goal of the TLO concept is to achieve uniqueness of keys in the model path portion of the object name. To come as close as possible to this goal, the TLO must have relevance in an enterprise context.

An object in the scope of another object can in turn be a scope for a different object. Therefore, all model object instances are arranged in directed graphs with the TLOs as peer roots. The structure of this graph, which defines which classes are in the scope of another given class, is part of CIM by means of associations qualified with the Weak qualifier.

8.2.1 Referencing Weak Objects

A reference to an instance of an association includes the propagated keys. The properties must have the propagated qualifier that identifies the class in which the property originates and the name of the property in that class. For example:

```
instance of Acme_has
{
   anOS = "Acme_OS.Name=\"acmeunit\",SystemName=\"UnixHost\"";
   aUser = "Acme_User.uid=33,OSName=\"acmeunit\",SystemName=\"UnixHost\"";
};
```

The operating system being weak to system is declared as follows:

```
Class Acme_OS
{
      [key]
   String Name;
      [key, Propagated("CIM_System.Name")]
   String SystemName;
};
```

The user class being weak to operating system is declared as follows:

```
Class Acme_User
{
      [key]
   String uid;
      [key, Propagated("Acme_OS.Name")]
   String OSName;
      [key, Propagated("Acme_OS.SystemName")]
   String SystemName;
};
```

8.3 Naming CIM Objects

Because CIM allows multiple implementations, it is not sufficient to think of the name of an object as just the combination of properties that have the Key qualifier. The name must also identify the implementation that actually hosts the objects. The object name consists of the namespace path, which provides access to a CIM implementation, plus the model path, which provides full navigation within the CIM schema. The namespace path is used to locate a particular namespace. The details of the namespace path depend on the implementation. The model path is the concatenation of the class name and the properties of the class that are qualified with the Key qualifier. When the class is weak with respect to another class, the model path includes all key properties from the scoping objects. Figure 11 shows the various components of an object name. These components are described in more detail in the following clauses. See the objectName non-terminal in ANNEX A for the formal description of object name syntax.

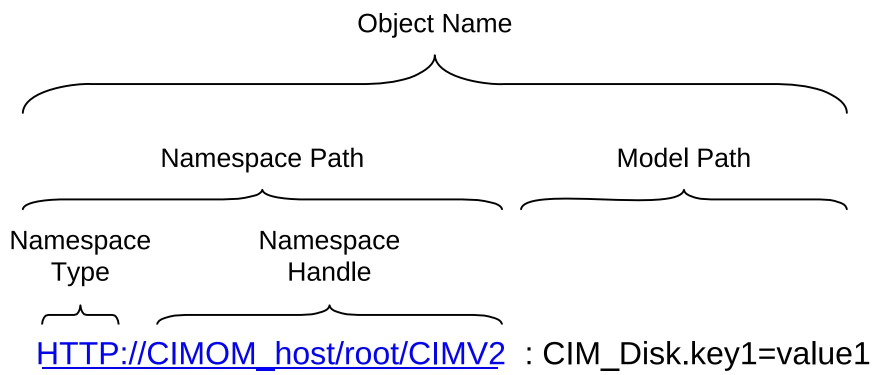


Figure 11 – Object Naming

8.3.1 Namespace Path

A namespace path references a namespace within an implementation that can host CIM objects. A namespace path resolves to a namespace hosted by a CIM-capable implementation (in other words, a CIM object manager). Unlike in the model path, the details of the namespace path are implementation-specific. Therefore, the namespace path identifies the following details:

- the implementation or namespace type
- a handle that references a particular implementation or namespace handle

8.3.1.1 Namespace Type

The namespace type classifies or identifies the type of implementation. The provider of the implementation must describe the access protocol for that implementation, which is analogous to specifying http or ftp in a browser.

Fundamentally, a namespace type implies an access protocol or API set to manipulate objects. These APIs typically support the following operations:

- generating a MOF file for a particular scope of classes and associations
- importing a MOF file
- manipulating instances

A particular management platform can access management information in a variety of ways. Each way must have a namespace type definition. Given this type, there is an assumed set of mechanisms for exporting, importing, and updating instances.

8.3.1.2 Namespace Handle

The namespace handle identifies a particular instance of the type of implementation. This handle must resolve to a namespace within an implementation. The details of the handle are implementation-specific. It might be a simple string for an implementation that supports one namespace, or it might be a hierarchical structure if an implementation supports multiple namespaces. Either way, it resolves to a namespace.

Some implementations can support multiple namespaces. In this case, the implementation-specific reference must resolve to a particular namespace within that implementation (see Figure 12).

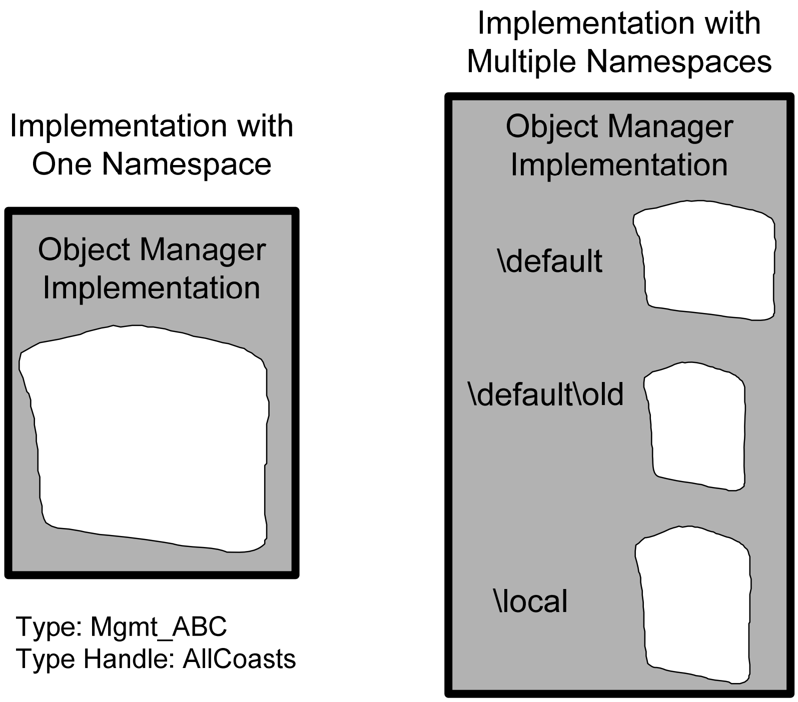


Figure 12 – Namespaces

Two important points to remember about namespaces are as follows:

- Namespaces can overlap with respect to their contents.
- When an object in one namespace has the same model path as an object in another namespace, this does not guarantee that the objects are representing the same reality.

8.3.2 Model Path

The object name constructed as a scoping path through the CIM schema is called a model path. A model path for an instance is a combination of the key property names and values qualified by the class name. It is solely described by CIM elements and is absolutely implementation-independent. It can describe the path to a particular object or to identify a particular object within a namespace. The name of any instance is a concatenation of named key property values, including all key values of its scoping objects. When the class is weak with respect to another class, the model path includes all key properties from the scoping objects.

The formal syntax of model path is provided in ANNEX A.

The syntax of model path is as follows:

<className>.<key1>=<value1>[,<keyx>=<valuex>]*

8.3.3 Specifying the Object Name

There are various ways to specify the object name details for any class instance or association reference in a MOF file.

The model path is specified differently for objects and associations. For objects (instances of classes), the model path is the combination of property value pairs marked with the Key qualifier. Therefore, the model path for the following example is: "ex_sampleClass.label1=9921,label2=8821". Because the order of the key properties is not significant, the model path can also be: "ex_sampleClass.label2=8821,label1=9921".

```
Class ex_sampleClass
{
      [key]
   uint32 labe11;
      [key]
   string label2;
   uint32 size;
   uint32 weight;
};


instance of ex_sampleClass
{
   label1 = 9921;
   label2 = "SampleLabel";
   size = 80;
   weight = 45
};


instance of ex_sampleClass
{
   label1 = 0121;
   label2 = "Component";
   size = 80;
   weight = 45
};
```

For associations, a model path specifies the value of a reference in an INSTANCE OF statement for an association. In the following composedof-association example, the model path "ex_sampleClass.label1=9921,label2=8821" references an instance of the ex_sampleClass that is playing the role of a composer:

```
   [Association ]
Class ex_composedof
{
   [key] composer REF ex_sampleClass;
   [key] component REF ex_sampleClass;
};
instance of ex_composedof
{
   composer = "ex_sampleClass.label1=9921,label2=\"SampleLabel\"";
   component = "ex_sampleClass.label1=0121,label2=\"Component\"";
}
```

An object path for the ex_composedof instance is as follows. Notice how double quote characters are handled:

```
ex_composedof.composer="ex_sampleClass.label1=9921,label2=\"SampleLabel\"",componen
t="ex_sampleClass.label1=0121,label2=\"Component\""
```

Even in the unusual case of a reference to an association, the object name is formed the same way:

```
    [Association]
Class ex_moreComposed
{
    composedof REF ex_composedof;
    . . .
};

instance of ex_moreComposed
{
    composedof =
    "ex_composedof.composer=\"ex_sampleClass.label1=9921,label2=\\\"SampleLabel\\\"
    \",component=\"ex_sampleClass.label1=0121,label2=\\\"Component\\\"\"";
    . . .
};
```

The object name can be used as the value for object references and for object queries.

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 13).

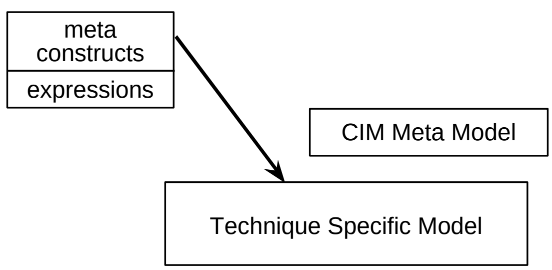


Figure 13 – Technique Mapping Example

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 14 illustrates the results.

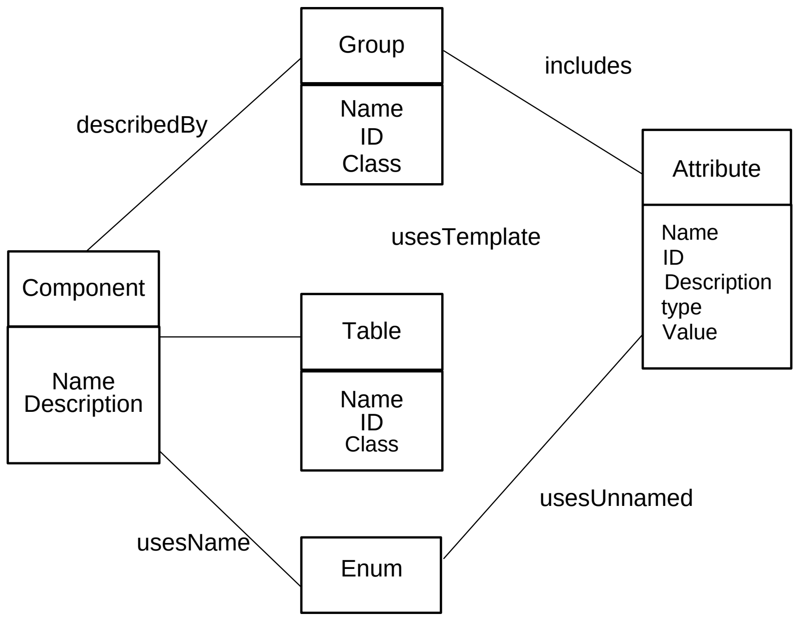


Figure 14 – MIF Technique Mapping Example

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 15). 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.

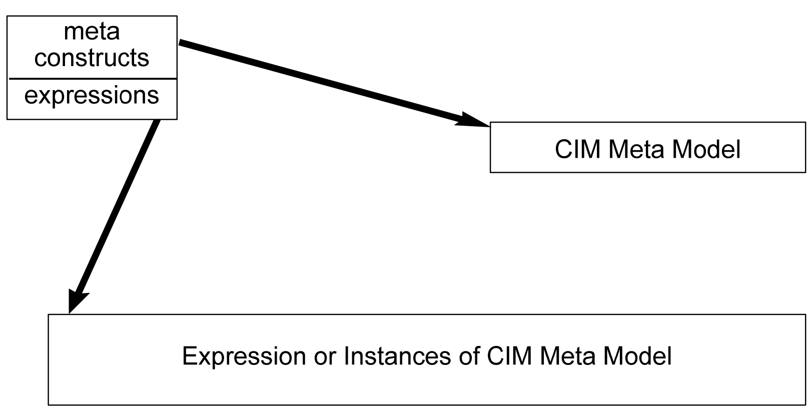


Figure 15 – Recast Mapping

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 8.

Table 8 – Domain Mapping Example

CIM "Disk" Property	Can Be Sourced from DMI Group/Attribute
StorageType	"MIF.DMTF|Disks|002.1"
StorageInterface	"MIF.DMTF|Disks|002.3"
RemovableDrive	"MIF.DMTF|Disks|002.6"
RemovableMedia	"MIF.DMTF|Disks|002.7"
DiskSize	"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 9, the repository can be organized into the four partitions: meta, technique, recast, and domain (see Figure 16).

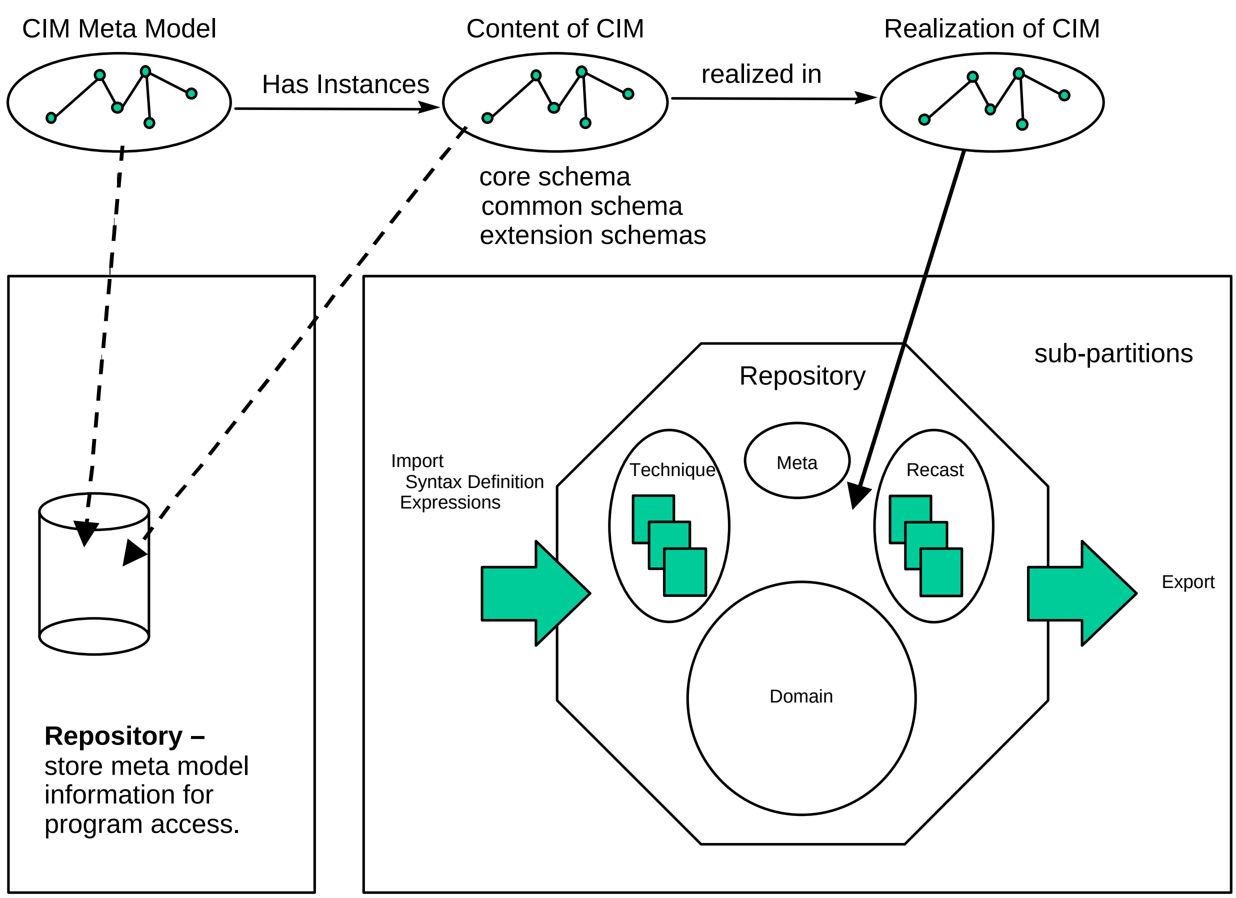


Figure 16 – Repository Partitions

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 9.4), 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 17 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.

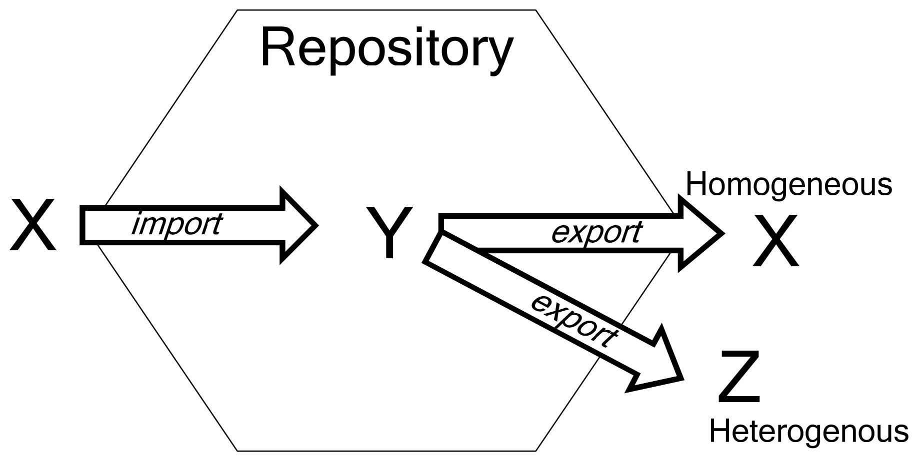


Figure 17 – Homogeneous and Heterogeneous Export

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 18).

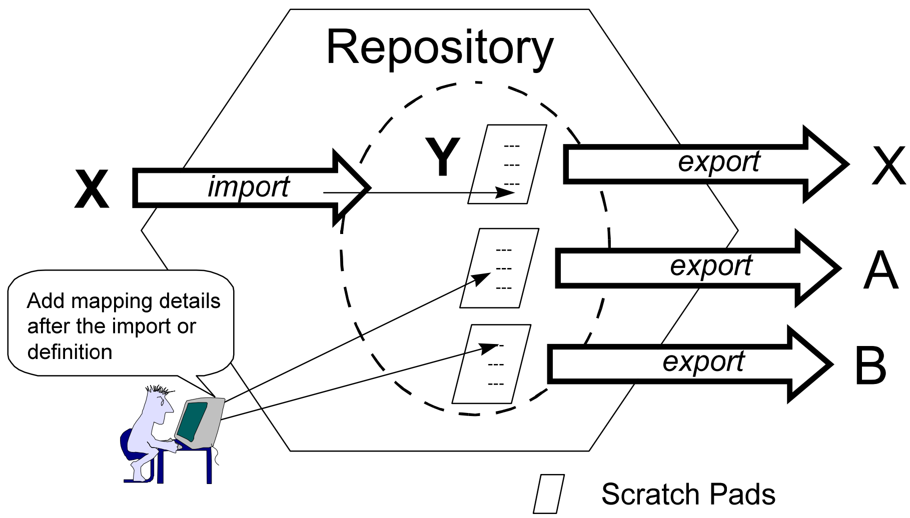


Figure 18 – Scratch Pads and Mapping

Figure 18 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, note these points:

1) An empty property list is equivalent to `"*"`.

2) All keywords are case-insensitive.

3) The IDENTIFIER type is used for names of classes, properties, qualifiers, methods, and namespaces. The rules governing the naming of classes and properties are presented in ANNEX E.

4) A string value may contain quote (`"`) characters, if each is immediately preceded by a backslash (`\`) character.

5) In the current release, the MOF BNF does not support initializing an array value to empty (an array with no elements). In the 3.0 version of this specification, the DMTF plans to extend the MOF BNF to support this functionality as follows:

   arrayInitialize = "{" [arrayElementList] "}"

   arrayElementList = constantValue *("," constantValue)

   To ensure interoperability with the V2.x implementations, 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 the 3.0 version or higher to initialize an empty array.

The following is the grammar for the MOF syntax:

```
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 "}" ";"

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 ]

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 |
                            booleanValue | nullValue

integerValue            =   binaryValue | octalValue | decimalValue | hexValue

referenceInitializer    =   objectHandle | aliasIdentifier

objectHandle            =   stringValue
                            // the(unescaped)contents of which must form an
                            // objectName; see examples

objectName                  [ namespacePath ":" ]  modelPath

namespacePath               [ namespaceType "://" ] namespaceHandle

namespaceType               One or more UCS-2 characters NOT including the sequence
                            "://"

namespaceHandle         =   One or more UCS-2 character, possibly including ":"
                            // Note that modelPath may also contain ":" characters
                            // within quotes; some care is required to parse
                            // objectNames.

modelPath               =   className "." keyValuePairList
                            // Note: className alone represents a path to a class,
                            // rather than an instance

keyValuePairList        =   keyValuePair *( "," keyValuePair )
```

```
keyValuePair            =  ( propertyName "=" constantValue ) | ( referenceName "="
                           objectHandle )

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 productions do not allow white space between the terms:

```
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               =  // any single-quoted Unicode-character, except
```

```
                            // single quotes

stringValue          =  1*( """ *stringChar """ )

stringChar           =  "\" """ | // encoding for double-quote
                        "\" "\" | // encoding for backslash
                        any UCS-2 character but """ or "\"

booleanValue         =  TRUE | FALSE

nullValue            =  NULL
```

3181 The remaining productions 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

```
// ==================================================================
//    NamedElement
// ==================================================================
        [Version("2.3.0"), Description(
        "The Meta_NamedElement class represents the root class for the "
        "Metaschema. It has one property: Name, which is inherited by all the "
        "non-association classes in the Metaschema. Every metaconstruct is "
        "expressed as a descendent of the class Meta_Named Element.") ]
class Meta_NamedElement
{
        [Description (
        "The Name property indicates the name of the current Metaschema element. "
        "The following rules apply to the Name property, depending on the "
        "creation type of the object:<UL><LI>Fully-qualified class names, such "
        "as those prefixed by the schema name, are unique within the schema."
        "<LI>Fully-qualified association and indication names are unique within "
        "the schema (implied by the fact that association and indication classes "
        "are subtypes of Meta_Class). <LI>Implicitly-defined qualifier names are "
        "unique within the scope of the characterized object; that is, a named "
        "element may not have two characteristics with the same name."
        "<LI>Explicitly-defined qualifier names are unique within the defining "
        "schema. An implicitly-defined qualifier must agree in type, scope and "
        "flavor with any explicitly-defined qualifier of the same name."
        "<LI>Trigger names must be unique within the property, class or method "
        "to which the trigger applies. <LI>Method and property names must be "
        "unique within the domain class. A class can inherit more than one "
        "property or method with the same name. Property and method names can be "
        "qualified using the name of the declaring class. <LI>Reference names "
        "must be unique within the scope of their defining association class. "
        "Reference names obey the same rules as property names. </UL><B>Note:</B> "
        "Reference names are not required to be unique within the scope of the "
        "related class. Within such a scope, the reference provides the name of "
        "the class within the context defined by the association.") ]
    string Name;
};

// ==================================================================
//    QualifierFlavor
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_QualifierFlavor class encapsulates extra semantics attached "
        "to a qualifier such as the rules for transmission from superClass "
        "to subClass and whether or not the qualifier value may be translated "
        "into other languages") ]
class Meta_QualifierFlavor:Meta_NamedElement
{
};
```

```
// ==================================================================
//    Schema
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_Schema class represents a group of classes with a single owner."
        " Schemas are used for administration and class naming. Class names must "
        "be unique within their owning schemas.") ]
class Meta_Schema:Meta_NamedElement
{
};

// ==================================================================
//    Trigger
// ==================================================================
        [Version("2.3.0"), Description (
        "A Trigger is a recognition of a state change (such as create, delete, "
        "update, or access) of a Class instance, and update or access of a "
        "Property.") ]
class Meta_Trigger:Meta_NamedElement
{
};

// ==================================================================
//    Qualifier
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_Qualifier class represents characteristics of named elements. "
        "For example, there are qualifiers that define the characteristics of a "
        "property or the key of a class. Qualifiers provide a mechanism that "
        "makes the Metaschema extensible in a limited and controlled fashion."
        "<P>It is possible to add new types of qualifiers by the introduction of "
        "a new qualifier name, thereby providing new types of metadata to "
        "processes that manage and manipulate classes, properties, and other "
        "elements of the Metaschema.") ]
class Meta_Qualifier:Meta_NamedElement
{
        [Description ("The Value property indicates the value of the qualifier.")]
    string Value;
};

// ==================================================================
//    Method
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Method class represents a declaration of a signature; that is, "
        "the method name, return type and parameters, and (in the case of a "
        "concrete class) may imply an implementation.") ]
class Meta_Method:Meta_NamedElement
{
};

// ==================================================================
//    Property
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Property class represents a value used to characterize "
        "instances of a class. A property can be thought of as a pair of Get and "
        "Set functions that, when applied to an object, return state and set "
        "state, respectively.") ]
class Meta_Property:Meta_NamedElement
{
};
```

```
// ==================================================================
//    Reference
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Reference class represents (and defines) the role each object "
        "plays in an association. The reference represents the role name of a "
        "class in the context of an association, which supports the provision of "
        "multiple relationship instances for a given object. For example, a "
        "system can be related to many system components.") ]
class Meta_Reference:Meta_Property
{
};

// ==================================================================
//    Class
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Class class is a collection of instances that support the same "
        "type; that is, the same properties and methods. Classes can be arranged "
        "in a generalization hierarchy that represents subtype relationships "
        "between classes. <P>The generalization hierarchy is a rooted, directed "
        "graph and does not support multiple inheritance. Classes can have "
        "methods, which represent the behavior relevant for that class. A Class "
        "may participate in associations by being the target of one of the "
        "references owned by the association.") ]
class Meta_Class:Meta_NamedElement
{
};

// ==================================================================
//    Indication
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Indication class represents an object created as a result of a "
        "trigger. Because Indications are subtypes of Meta_Class, they can have "
        "properties and methods, and be arranged in a type hierarchy. ") ]
class Meta_Indication:Meta_Class
{
};

// ==================================================================
//    Association
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Association class represents a class that contains two or more "
        "references and represents a relationship between two or more objects. "
        "Because of how associations are defined, it is possible to establish a "
        "relationship between classes without affecting any of the related "
        "classes.<P>For example, the addition of an association does not affect "
        "the interface of the related classes; associations have no other "
        "significance. Only associations can have references. Associations can "
        "be a subclass of a non-association class. Any subclass of "
        "Meta_Association is an association.") ]
class Meta_Association:Meta_Class
{
};
```

```
// ==================================================================
//    Characteristics
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_Characteristics class relates a Meta_NamedElement to a "
        "qualifier that characterizes the named element. Meta_NamedElement may "
        "have zero or more characteristics.") ]
class Meta_Characteristics
{
        [Description (
        "The Characteristic reference represents the qualifier that "
        "characterizes the named element.") ]
    Meta_Qualifier REF Characteristic;
        [Aggregate, Description (
        "The Characterized reference represents the named element that is being "
        "characterized.") ]
    Meta_NamedElement REF Characterized;
};

// ==================================================================
//    PropertyDomain
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_PropertyDomain class represents an association between a class "
        "and a property.<P>A property  has only one domain: the class that owns "
        "the property. A property can have an override relationship with another "
        "property from a different class. The domain of the overridden property "
        "must be a supertype of the domain of the overriding property. The "
        "domain of a reference must be an association.") ]
class Meta_PropertyDomain
{
        [Description (
        "The Property reference represents the property that is owned by the "
        "class referenced by Domain.") ]
    Meta_Property REF Property;
        [Aggregate, Description (
        "The Domain reference represents the class that owns the property "
        "referenced by Property.") ]
    Meta_Class REF Domain;
};

// ==================================================================
//    MethodDomain
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_MethodDomain class represents an association between a class "
        "and a method.<P>A method has only one domain: the class that owns the "
        "method, which can have an override relationship with another method "
        "from a different class. The domain of the overridden method must be a "
        "supertype of the domain of the overriding method. The signature of the "
        "method (that is, the name, parameters and return type) must be "
        "identical.") ]
class Meta_MethodDomain
{
        [Description (
        "The Method reference represents the method that is owned by the class "
        "referenced by Domain.") ]
    Meta_Method REF Method;
        [Aggregate, Description (
        "The Domain reference represents the class that owns the method "
        "referenced by Method.") ]
    Meta_Class REF Domain;
};
```

```
// ==================================================================
//    ReferenceRange
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_ReferenceRange class defines the type of the reference.") ]
class Meta_ReferenceRange
{
        [Description (
        "The Reference reference represents the reference whose type is defined "
        "by Range.") ]
    Meta_Reference REF Reference;
        [Description (
        "The Range reference represents the class that defines the type of "
        "reference.") ]
    Meta_Class REF Range;
};

// ==================================================================
//    QualifiersFlavor
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_QualifiersFlavor class represents an association between a "
        "flavor and a qualifier.") ]
class Meta_QualifiersFlavor
{
        [Description (
        "The Flavor reference represents the qualifier flavor to "
        "be applied to Qualifier.") ]
    Meta_QualifierFlavor REF Flavor;
        [Aggregate, Description (
        "The Qualifier reference represents the qualifier to which "
        "Flavor applies.") ]
    Meta_Qualifier REF Qualifier;
};

// ==================================================================
//    SubtypeSupertype
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_SubtypeSupertype class represents subtype/supertype "
        "relationships between classes arranged in a generalization hierarchy. "
        "This generalization hierarchy is a rooted, directed graph and does not "
        "support multiple inheritance.") ]
class Meta_SubtypeSupertype
{
        [Description (
        "The SuperClass reference represents the class that is hierarchically "
        "immediately above the class referenced by SubClass.") ]
    Meta_Class REF SuperClass;
        [Description (
        "The SubClass reference represents the class that is the immediate "
        "descendent of the class referenced by SuperClass.") ]
    Meta_Class REF SubClass;
};
```

```
// ==================================================================
//    PropertyOverride
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_PropertyOverride class represents an association between two "
        "properties where one overrides the other.<P>Properties have reflexive "
        "associations that represent property overriding. A property can "
        "override an inherited property, which implies that any access to the "
        "inherited property will result in the invocation of the implementation "
        "of the overriding property. A Property can have an override "
        "relationship with another property from a different class.<P>The domain "
        "of the overridden property must be a supertype of the domain of the "
        "overriding property. The class referenced by the Meta_ReferenceRange "
        "association of an overriding reference must be the same as, or a "
        "subtype of, the class referenced by the Meta_ReferenceRange "
        "associations of the reference being overridden.") ]
class Meta_PropertyOverride
{
        [Description (
        "The OverridingProperty reference represents the property that overrides "
        "the property referenced by OverriddenProperty.") ]
    Meta_Property REF OverridingProperty;
        [Description (
        "The OverriddenProperty reference represents the property that is "
        "overridden by the property reference by OverridingProperty.") ]
    Meta_Property REF OverriddenProperty;
};

// ==================================================================
//    MethodOverride
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_MethodOverride class represents an association between two "
        "methods, where one overrides the other. Methods have reflexive "
        "associations that represent method overriding. A method can override an "
        "inherited method, which implies that any access to the inherited method "
        "will result in the invocation of the implementation of the overriding "
        "method.") ]
class Meta_MethodOverride
{
        [Description (
        "The OverridingMethod reference represents the method that overrides the "
        "method referenced by OverriddenMethod.") ]
    Meta_Method REF OverridingMethod;
        [Description (
        "The OverriddenMethod reference represents the method that is overridden "
        "by the method reference by OverridingMethod.") ]
    Meta_Method REF OverriddenMethod;
};

// ==================================================================
//    ElementSchema
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_ElementSchema class represents the elements (typically classes "
        "and qualifiers) that make up a schema.") ]
class Meta_ElementSchema
{
        [Description (
        "The Element reference represents the named element that belongs to the "
        "schema referenced by Schema.") ]
    Meta_NamedElement REF Element;
        [Aggregate, Description (
```

```
            "The Schema reference represents the schema to which the named element "
            "referenced by Element belongs.") ]
        Meta_Schema REF Schema;
    };
```

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. Note that 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 production "programmatic-unit" in the syntax defined in this annex. This syntax supports any type of unit, including SI units, United States units, and any other standard or non-standard units. The syntax definition here uses ABNF with the following exceptions:

- Rules separated by a bar (|) represent choices (instead of using a forward slash (/) as defined in ABNF).
- Any characters must be processed case sensitively instead of case-insensitively, as defined in ABNF.

ABNF defines the items in the syntax as assembled without inserted white space. Therefore, the syntax explicitly specifies any white space. The ABNF syntax is defined as follows:

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 = ( "A"..."Z" | "a"..."z" | "_" | U+0080...U+FFEF )
UNIT-CHAR = ( FIRST-UNIT-CHAR | "0"..."9" | "-" )
ZERO = "0"
NON-ZERO-DIGIT = ("1"..."9")
DIGIT = ZERO | NON-ZERO-DIGIT
WS = ( S | TAB | NL )
S = U+0020
TAB = U+0009
NL = U+000A
```

Unicode characters used in the syntax:

```
U+0009 = "\t"      (tab)
U+000A = "\n"      (newline)
U+0020 = " "       (space)
U+0080...U+FFEF = (other Unicode characters)
```

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 production "base-unit" 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. Note that "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) 1 dB = 20 · lg (U/U0)	Logarithmic ratio (dimensionless unit) Used with a factor of 10 for power, intensity, and so on. Used with a factor of 20 for voltage, pressure, loudness of sound, and so on
count			Unit for counted items or phenomenons. The description of the schema element using this unit should describe what kind of item or phenomenon is counted.
revolution	rev	1 rev = 360°	Turn, plane angle
degree	°	180° = pi rad	Plane angle
radian	rad	1 rad = 1 m/m	Plane angle
steradian	sr	1 sr = 1 m^2/m^2	Solid angle
bit	bit		Quantity of information
byte	B	1 B = 8 bit	Quantity of information
dataword	word	1 word = N bit	Quantity of information. The number of bits depends on the computer architecture.
meter	m	SI base unit	Length (The corresponding ISO SI unit is "metre.")
inch	in	1 in = 0.0254 m	Length
rack unit	U	1 U = 1.75 in	Length (height unit used for computer components, as defined in EIA-310)
foot	ft	1 ft = 12 in	Length
yard	yd	1 yd = 3 ft	Length
mile	mi	1 mi = 1760 yd	Length (U.S. land mile)
liter	l	1000 l = 1 m^3	Volume (The corresponding ISO SI unit is "litre.")
fluid ounce	fl.oz	33.8140227 fl.oz = 1 l	Volume for liquids (U.S. fluid ounce)
liquid gallon	gal	1 gal = 128 fl.oz	Volume for liquids (U.S. liquid gallon)
mole	mol	SI base unit	Amount of substance
kilogram	kg	SI base unit	Mass
ounce	oz	35.27396195 oz = 1 kg	Mass (U.S. ounce, avoirdupois ounce)

PU Base Unit	Symbol	Calculation	Quantity
pound	lb	1 lb = 16 oz	Mass (U.S. pound, avoirdupois pound)
second	s	SI base unit	Time
minute	min	1 min = 60 s	Time
hour	h	1 h = 60 min	Time
day	d	1 d = 24 h	Time
week	week	1 week = 7 d	Time
hertz	Hz	1 Hz = 1 /s	Frequency
gravity	g	1 g = 9.80665 m/s²	Acceleration
degree celsius	°C	1 °C = 1 K (diff)	Thermodynamic temperature
degree fahrenheit	°F	1 °F = 5/9 K (diff)	Thermodynamic temperature
kelvin	K	SI base unit	Thermodynamic temperature, color temperature
candela	cd	SI base unit	Luminous intensity
lumen	lm	1 lm = 1 cd·sr	Luminous flux
nit	nit	1 nit = 1 cd/m²	Luminance
lux	lx	1 lx = 1 lm/m²	Illuminance
newton	N	1 N = 1 kg·m/s²	Force
pascal	Pa	1 Pa = 1 N/m²	Pressure
bar	bar	1 bar = 100000 Pa	Pressure
decibel(A)	dB(A)	1 dB(A) = 20 lg (p/p0)	Loudness of sound, relative to reference sound pressure level of p0 = 20 µPa in gases, using frequency weight curve (A)
decibel(C)	dB(C)	1 dB(C) = 20 · lg (p/p0)	Loudness of sound, relative to reference sound pressure level of p0 = 20 µPa in gases, using frequency weight curve (C)
joule	J	1 J = 1 N·m	Energy, work, torque, quantity of heat
watt	W	1 W = 1 J/s	Power, radiant flux
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

PU Base Unit	Symbol	Calculation	Quantity
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. For displaying a unit, the client application should construct the string to be displayed from the PUnit qualifier using the conventions of the client application.

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[3]
- Amps at <Numeric Value> Volts
- Clock Ticks
- Packets, per Thousand Packets

[3] Standard Rack Measurement equal to 1.75 inches.

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.

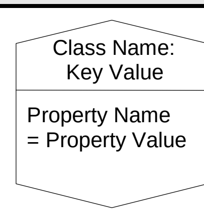
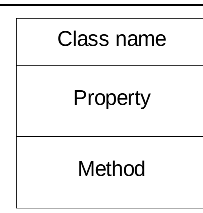
A line decorated with a triangle indicates an inheritance relationship; the lower rectangle represents a subtype of the upper rectangle. The triangle points to the superclass.

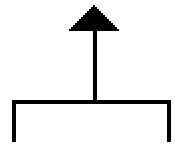
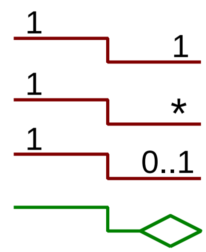
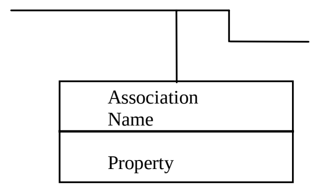
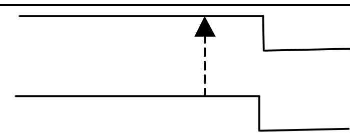
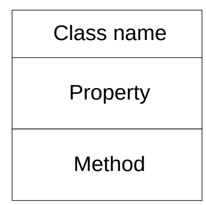
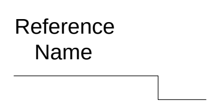
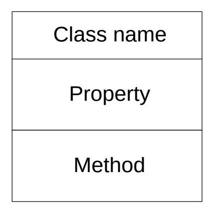
Other solid lines represent relationships. The cardinality of the references on either side of the relationship is indicated by a decoration on either end. The following character combinations are commonly used:

- "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		

Meta Element	Interpretation	Diagramming Notation
Subclass		
Association	1:1 1:Many 1:zero or 1 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 **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	

Meta Element	Interpretation	Diagramming Notation
Trigger	State transition diagrams	
Qualifier	No direct equivalent	

ANNEX E
(normative)

Unicode Usage

All punctuation symbols associated with object path or MOF syntax occur within the Basic Latin range U+0000 to U+007F. These symbols include normal punctuators, such as slashes, colons, commas, and so on. No important syntactic punctuation character occurs outside of this range.

All characters above U+007F are treated as parts of names, even though there are several reserved characters such as U+2028 and U+2029, which are logically white space. Therefore, all namespace, class, and property names are identifiers composed as follows:

- Initial identifier characters must be in set S1, where S1 = {U+005F, U+0041...U+005A, U+0061...U+007A, U+0080...U+FFEF) (This includes alphabetic characters and the underscore.)
- All following characters must be in set S2 where S2 = S1 union {U+0030...U+0039} (This includes alphabetic characters, Arabic numerals 0 through 9, and the underscore.)

Note that the Unicode specials range (U+FFF0...U+FFFF) are not legal for identifiers. While the preceding sub-range of U+0080...U+FFEF includes many diacritical characters that would not be useful in an identifier, as well as the Unicode reserved sub-range that is not allocated, it seems advisable for simplicity of parsers simply to treat this entire sub-range as legal for identifiers.

Refer to RFC2279 for an example of a Universal Transformation Format with specific characteristics for dealing with multi-octet characters on an application-specific basis.

E.1 MOF Text

MOF files using Unicode must contain a signature as the first two bytes of the text file, either U+FFFE or U+FEFF, depending on the byte ordering of the text file (as suggested in Section 2.4 of the ISO/IEC 10646:2003). U+FFFE is little endian.

All MOF keywords and punctuation symbols are as described in the MOF syntax document and are not locale-specific. They are composed of characters falling in the range U+0000...U+007F, regardless of the locale of origin for the MOF or its identifiers.

E.2 Quoted Strings

In all cases where non-identifier string values are required, delimiters must surround them. The supported delimiter for strings is U+0027. When a quoted string is started using the delimiter, the same delimiter, U+0027, is used to terminate it. In addition, the digraph U+005C ("\") followed by U+0027 "'" constitutes an embedded quotation mark, not a termination of the quoted string. The characters permitted within these quotation mark delimiters may fall within the range U+0001 through U+FFEF.

ANNEX F
(informative)

Guidelines

The following are guidelines for modeling:

- Method descriptions are recommended and must, at a minimum, indicate the method's side effects (pre- and post-conditions).
- Associations must not be declared as subtypes of classes that are not associations.
- 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. This qualifier must *always* be present for string properties used as keys.
- A class with no Abstract qualifier must define, or inherit, key properties.

F.1 Mapping of Octet Strings

Most management models, including SNMP and DMI, support octet strings as data types. The octet string data type represents arbitrary numeric or textual data that is stored as an indexed byte array of unlimited but fixed size. Typically, the first n bytes indicate the actual string length. Because some environments reserve only the first byte, they do not support octet strings larger than 255 bytes.

In the current release, CIM does not support octet strings as a separate data type. To map a single octet string (that is, an octet of binary data), the equivalent CIM property should be defined as an array of unsigned 8-bit integers (uint8). The first four bytes of the array contain the length of the octet data: byte 0 is the most significant byte of the length, and byte 3 is the least significant byte. The octet data starts at byte 4. The OctetString qualifier may be used to indicate that the uint8 array conforms to this encoding.

Arrays of uint8 arrays are not supported. Therefore, to map an array of octet strings, a textual convention encoding the binary information as hexadecimal digit characters (such as 0x<&lt0-9,A-F>&lt0-9,A-F>>*) is used for each octet string in the array. The number of octets in the octet string is encoded in the first 8 hexadecimal digits of the string with the most significant digits in the left-most characters of the string. The length count octets are included in the length count. For example, "0x00000004" is the encoding of a 0-length octet string.

The OctetString qualifier qualifies the string array.

EXAMPLE: Example use of the OctetString qualifier on a property is as follows:

```
[Description ("An octet string"), Octetstring]
uint8 Foo[];
[Description ("An array of octet strings"), Octetstring]
String Bar[];
```

F.2 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 sql1992.txt (ANNEX E):

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			

3777 From sql3part2.txt (ANNEX E):

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		

3778 sql3part4.txt (ANNEX E):

CALL	DO	ELSEIF	EXCEPTION
IF	LEAVE	LOOP	OTHERS
RESIGNAL	RETURN	RETURNS	SIGNAL
TUPLE	WHILE		

ANNEX G
(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. Clients that do not handle embedded objects may treat properties with this qualifier just like any other string-valued property. 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.

G.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 nonterminals, as appropriate in embedded classes (see ANNEX A).

EXAMPLE:

```
Instance of CIM_InstCreation {
   EventTime = "20000208165854.457000-360";
   SourceInstance =
      "Instance of CIM_FAN {"
      "DeviceID = \"Fan 1\";"
      "Status = \"Degraded\";"
      "};";
};
Instance of CIM_ClassCreation {
   EventTime = "20031120165854.457000-360";
   ClassDefinition =
      "class CIM_Fan : CIM_CoolingDevice {"
      " boolean VariableSpeed;"
      "   [Units (\"Revolutions per Minute\") ]"
      "uint64 DesiredSpeed;"
      "};"
};
```

G.2 Encoding for CIM-XML

When the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance are rendered in CIM-XML, the embedded object must be encoded into string form as either an INSTANCE element (for instances) or a CLASS element (for classes), as defined in the DMTF DSP0200, and DSP0201.

ANNEX H
(informative)

Schema Errata

Based on the concepts and constructs in this specification, 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 5.3.
- 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 5.5.2.11.

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 I
(informative)

Ambiguous Property and Method Names

In 5.1, item 21)-e) explicitly allows 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 applications to be written under the assumption that property names alone suffice to identify properties uniquely. However, such applications 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;
};
```

If there is just one instance of VENDOR_Subclass, a call to enumerateInstances("VENDOR_Subclass") might produce the following XML result from the CIMOM if it did not bother to ask for CLASSORIGIN information:

```
<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 call 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 client application 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 originClass 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 an application 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;
};
```

An application 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 XML from a CIMOM 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 client application 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, the application 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 server 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 J
(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 sections 7.5.4 "Navigation to Association Classes" and 7.5.5 "Navigation from Association Classes". Note that 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.a12.r.p2"}]
// Using a12 is required to disambiguate end name r
class C1 {
  string p1;
};
[ClassConstraint {
  "inv i2: self.p2 = self.a12.x.p1",  // Using a12 is recommended
  "inv i3: self.p2 = self.x.p1"}]     // Works, but not recommended
class C2 {
  string p2;
};
```

```
class C3 { };
[Association] class A12 {
  C1 REF x;
  C2 REF r;  // same name as A13::r
};
[Association] class A13 {
  C1 REF y;
  C3 REF r;  // same name as A12::r
};
```

ANNEX K
(informative)

Change Log

Version	Date	Description
2.5.0a	2008/04/22	Initial creation – this version incorporates the ISO edits
2.5.0b	2009/02/16	Incorporated ArchCR0129, ArchCR0130
2.5.0c	2009/02/27	Comment resolution on WG ballot
2.5.0	2009/05/01	DMTF Standard Release

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 (1975) An Introduction to General Systems Thinking (1975 ed., Wiley-Interscience) (2001 ed. Dorset House).

Unicode Consortium, *The Unicode Standard*, Version 2.0, Addison-Wesley, 1996.