



1

2

3

4

Document Number: DSP0004

Date: 2009-05-01

Version: 2.5.0

5

Common Information Model (CIM) Infrastructure

6

Document Type: Specification

7

Document Status: DMTF Standard

8

Document Language: E

9

10 Copyright notice

11 Copyright © 2009 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

12 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems
13 management and interoperability. Members and non-members may reproduce DMTF specifications and
14 documents, provided that correct attribution is given. As DMTF specifications may be revised from time to
15 time, the particular version and release date should always be noted.

16 Implementation of certain elements of this standard or proposed standard may be subject to third party
17 patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations
18 to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose,
19 or identify any or all such third party patent right, owners or claimants, nor for any incomplete or
20 inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to
21 any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize,
22 disclose, or identify any such third party patent rights, or for such party's reliance on the standard or
23 incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any
24 party implementing such standard, whether such implementation is foreseeable or not, nor to any patent
25 owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is
26 withdrawn or modified after publication, and shall be indemnified and held harmless by any party
27 implementing the standard from any and all claims of infringement by a patent owner for such
28 implementations.

29 For information about patents held by third-parties which have notified the DMTF that, in their opinion,
30 such patent may relate to or impact implementations of DMTF standards, visit
31 <http://www.dmtf.org/about/policies/disclosures.php>.

32

CONTENTS

34 Foreword 5

35 Introduction 6

36 1 Scope 9

37 2 Normative References..... 9

38 2.1 Approved References 10

39 2.2 Other References..... 11

40 3 Terms and Definitions 11

41 3.1 Keywords 11

42 3.2 Terms 11

43 4 Symbols and Abbreviated Terms 15

44 5 Meta Schema 16

45 5.1 Definition of the Meta Schema..... 17

46 5.2 Data Types..... 22

47 5.3 Supported Schema Modifications 28

48 5.4 Class Names..... 29

49 5.5 Qualifiers 30

50 6 Managed Object Format..... 55

51 6.1 MOF Usage..... 56

52 6.2 Class Declarations 56

53 6.3 Instance Declarations 56

54 7 MOF Components 56

55 7.1 Keywords 56

56 7.2 Comments..... 56

57 7.3 Validation Context..... 57

58 7.4 Naming of Schema Elements 57

59 7.5 Class Declarations 57

60 7.6 Association Declarations 62

61 7.7 Qualifier Declarations..... 64

62 7.8 Instance Declarations 65

63 7.9 Method Declarations 68

64 7.10 Compiler Directives..... 69

65 7.11 Value Constants..... 69

66 7.12 Initializers 71

67 8 Naming 72

68 8.1 Background..... 73

69 8.2 Weak Associations: Supporting Key Propagation 76

70 8.3 Naming CIM Objects 77

71 9 Mapping Existing Models into CIM 81

72 9.1 Technique Mapping 81

73 9.2 Recast Mapping 82

74 9.3 Domain Mapping..... 84

75 9.4 Mapping Scratch Pads..... 85

76 10 Repository Perspective 85

77 10.1 DMTF MIF Mapping Strategies..... 86

78 10.2 Recording Mapping Decisions 87

79 ANNEX A (normative) MOF Syntax Grammar Description..... 90

80 ANNEX B (informative) CIM Meta Schema 95

81 ANNEX C (normative) Units..... 102

82 C.1 Programmatic Units 102

83 C.2 Value for Units Qualifier 106

84 ANNEX D (informative) UML Notation 108

85	ANNEX E (normative) Unicode Usage	111
86	E.1 MOF Text	111
87	E.2 Quoted Strings	111
88	ANNEX F (informative) Guidelines	112
89	F.1 Mapping of Octet Strings	112
90	F.2 SQL Reserved Words	113
91	ANNEX G (normative) EmbeddedObject and EmbeddedInstance Qualifiers	115
92	G.1 Encoding for MOF	115
93	G.2 Encoding for CIM-XML	116
94	ANNEX H (informative) Schema Errata	117
95	ANNEX I (informative) Ambiguous Property and Method Names	119
96	ANNEX J (informative) OCL Considerations	122
97	ANNEX K (informative) Change Log	124
98	Bibliography	125
99		

100 Figures

101	Figure 1 – Four Ways to Use CIM	7
102	Figure 2 – Meta Schema Structure	19
103	Figure 3 – Reference Naming	20
104	Figure 4 – References, Ranges, and Domains	21
105	Figure 5 – References, Ranges, Domains, and Inheritance	21
106	Figure 6 – Example for Mapping a String Format Based on the General Mapping String Format	54
107	Figure 7 – Definitions of Instances and Classes	73
108	Figure 8 – Exporting to MOF	74
109	Figure 9 – Information Exchange	75
110	Figure 10 – Example of Weak Association	76
111	Figure 11 – Object Naming	78
112	Figure 12 – Namespaces	79
113	Figure 13 – Technique Mapping Example	81
114	Figure 14 – MIF Technique Mapping Example	82
115	Figure 15 – Recast Mapping	82
116	Figure 16 – Repository Partitions	85
117	Figure 17 – Homogeneous and Heterogeneous Export	87
118	Figure 18 – Scratch Pads and Mapping	88
119		

120 Tables

121	Table 1 – Standards Bodies	9
122	Table 2 – Intrinsic Data Types	22
123	Table 3 – Changes that Increment the CIM Schema Major Version Number	29
124	Table 4 – Meta Qualifiers	30
125	Table 5 – Recognized Flavor Types	61
126	Table 6 – UML Cardinality Notations	64
127	Table 7 – Standard Compiler Directives	69
128	Table 8 – Domain Mapping Example	84
129	Table C-1 – Base Units for Programmatic Units	104
130	Table D-1 – Diagramming Notation and Interpretation Summary	108
131		

132

Foreword

133 The *Common Information Model (CIM) Infrastructure* (DSP0004) was prepared by the DMTF Architecture
134 Working Group.

135 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems
136 management and interoperability.

137 Throughout this document, elements of formal syntax are described in the notation defined in [RFC 4234](#),
138 with these deviations:

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

143 The DMTF acknowledges the following people.

144 Editor:

- 145 • Lawrence Lamers – VMware

146 Contributors:

- 147 • Jeff Piazza – HP
- 148 • Andreas Maier – IBM
- 149 • George Ericson – EMC
- 150 • Jim Davis – WBEM Solutions
- 151 • Karl Schopmeyer – Inova Development
- 152 • Steve Hand – Symantec

153

Introduction

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

162 **CIM Management Schema**

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

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

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

171 **Core Model**

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

176 **Common Model**

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

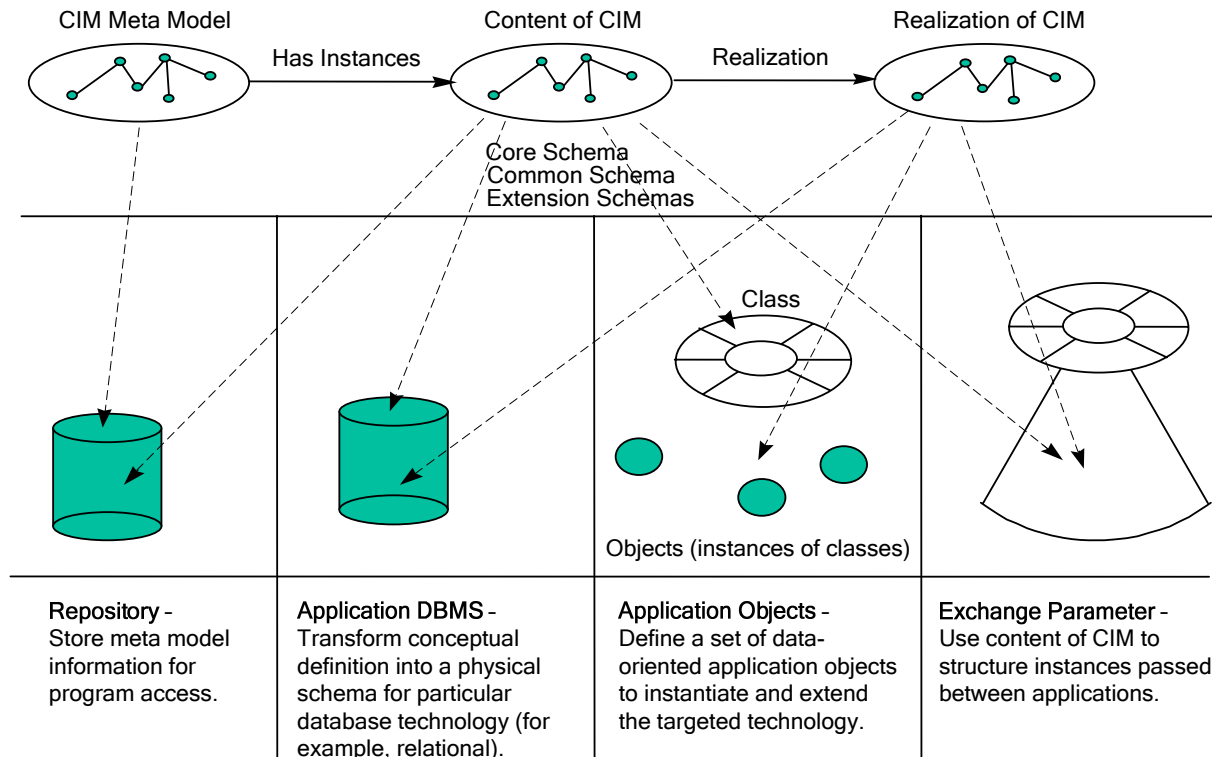
183 The common model is an information model common to particular management areas but independent of
184 a particular technology or implementation. The common areas are systems, applications, networks, and
185 devices. The information model is specific enough to provide a basis for developing management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

189 **Extension Schema**

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

193 **CIM Implementations**

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



197

198

Figure 1 – Four Ways to Use CIM

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

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

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

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

215 CIM Implementation Conformance

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

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

224 Trademarks

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

227

¹ The standard CIM application programming interface and/or communication protocol will be defined in a future version of the CIM Infrastructure specification.

228

Common Information Model (CIM) Infrastructure

229 1 Scope

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

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

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

240 2 Normative References

241 The following referenced documents are indispensable for the application of this document. For dated
 242 references, only the edition cited applies. For undated references, the latest edition of the referenced
 243 document (including any amendments) applies.

244 Copies of the following documents may be obtained from ANSI:

- 245 a) approved ANSI standards;
- 246 b) approved and draft international and regional standards (e.g., ISO, IEC); and
- 247 c) approved and draft foreign standards (e.g., JIS and DIN).

248 For further information, contact ANSI Customer Service Department at 212-642-4900 (phone), 212-302-
 249 1286 (fax) or via the World Wide Web at <http://www.ansi.org>.

250 Additional availability contact information is provided below as needed.

251 Table 1 shows standards bodies and their web sites.

252

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

253 **2.1 Approved References**

- 254 [ANSI/IEEE Standard 754-1985](#), *IEEE® Standard for Binary Floating-Point Arithmetic*, Institute of
255 Electrical and Electronics Engineers, August 1985.
- 256 CCITT [X.680](#) (07/02) *Information technology – Abstract Syntax Notation One (ASN.1): Specification of*
257 *basic notation*
- 258 DMTF [DSP0200](#), *CIM Operations over HTTP*, Version 1.3
- 259 DMTF [DSP4004](#), *DMTF Release Process*, Version 2.1
- 260 DMTF [DSP0201](#), *Specification for the Representation of CIM in XML*, Version 2.3
- 261 [EIA-310](#) Cabinets, Racks, Panels, and Associated Equipment
- 262 ISO 639-1:2002 *Codes for the representation of names of languages – Part 1: Alpha-2 code*
- 263 ISO 639-2:1998 *Codes for the representation of names of languages – Part 2: Alpha-3 code*
- 264 ISO 639-3:2007 *Codes for the representation of names of languages – Part 3: Alpha-3 code for*
265 *comprehensive coverage of languages*
- 266 ISO 1000:1992 *SI units and recommendations for the use of their multiples and of certain other units*
- 267 ISO 3166-1:2006 *Codes for the representation of names of countries and their subdivisions – Part 1:*
268 *Country codes*
- 269 ISO 3166-2:2007 *Codes for the representation of names of countries and their subdivisions – Part 2:*
270 *Country subdivision code*
- 271 ISO 3166-3:1999 *Codes for the representation of names of countries and their subdivisions – Part 3:*
272 *Code for formerly used names of countries*
- 273 ISO 8601:2004 (E), *Data elements and interchange formats – Information interchange – Representation*
274 *of dates and times*
- 275 ISO/IEC 9075-10:2003 *Information technology – Database languages – SQL – Part 10: Object Language*
276 *Bindings (SQL/OLB)*
- 277 ISO/IEC 10165-4:1992 *Information technology – Open Systems Interconnection – Structure of*
278 *management information – Part 4: Guidelines for the definition of managed objects (GDMO)*
- 279 ISO/IEC 10646:2003 *Information technology – Universal Multiple-Octet Coded Character Set (UCS)*
- 280 ISO/IEC 14750:1999 *Information technology – Open Distributed Processing – Interface Definition*
281 *Language*
- 282 ITU X.501: [Information Technology – Open Systems Interconnection – The Directory: Models](#)
- 283 OMG, [Object Constraint Language Version 2.0](#)
- 284 OMG, [UML Superstructure Specification, Version 2.1.1](#)
- 285 OMG, [UML Infrastructure Specification, Version 2.1.1](#)
- 286 OMG, [UML OCL Specification, Version 2.0](#)

287 2.2 Other References

- 288 ISO/IEC Directives, Part 2, [Rules for the structure and drafting of International Standards](#)
- 289 IETF, [RFC 2068](#), *Hypertext Transfer Protocol – HTTP/1.1*
- 290 IETF, [RFC 1155](#), *Structure and Identification of Management Information for TCP/IP-based Internets*
- 291 IETF, [RFC 2253](#), *Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names*
- 292
- 293 IETF, [RFC 2279](#), *UTF-8, a transformation format of ISO 10646*
- 294 IETF, [RFC 4234](#), *Augmented BNF for Syntax Specifications: ABNF*, 2005

295 3 Terms and Definitions

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

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

300 3.1 Keywords

301 3.1.1

302 **conditional**

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

305 3.1.2

306 **mandatory**

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

309 3.1.3

310 **optional**

311 indicates a course of action permissible within the limits of the document

312 3.1.4

313 **unspecified**

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

315 3.2 Terms

316 3.2.1

317 **aggregation**

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

322 **3.2.2**323 **association**

324 A [class](#) that expresses the relationship between two other *classes*. The relationship is established by two
325 or more [references](#) in the *association class* pointing to the related *classes*.

326 **3.2.3**327 **cardinality**

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

330 **3.2.4**331 **Common Information Model**332 **CIM**

333 Common Information Model is the schema of the overall managed environment. It is divided into a [core](#)
334 [model](#), [common model](#), and [extended schemas](#).

335 **3.2.5**336 **CIM schema**

337 The schema representing the [core](#) and [common models](#). The DMTF releases versions of this schema
338 over time as the schema evolves.

339 **3.2.6**340 **class**

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

344 **3.2.7**345 **common model**

346 A collection of [models](#) specific to a particular area and derived from the [core model](#). Included are the
347 system *model*, the application *model*, the network *model*, and the device *model*.

348 **3.2.8**349 **core model**

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

353 **3.2.9**354 **domain**

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

356 **3.2.10**357 **explicit qualifier**

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

362 **3.2.11**363 **extended schema**

364 A platform-specific [schema](#) derived from the common model. An example is the Win32 *schema*.

365 **3.2.12**366 **feature**

367 A [property](#) or [method](#) belonging to a *class*.

368 **3.2.13**369 **flavor**

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

373 **3.2.14**374 **implicit qualifier**

375 A [qualifier](#) that is a part of the definition of a [class](#), [property](#), or other schema element (see [explicit](#)
376 [qualifier](#)).

377 **3.2.15**378 **indication**

379 A type of [class](#) usually created as a result of a [trigger](#).

380 **3.2.16**381 **inheritance**

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

385 **3.2.17**386 **instance**

387 A unit of data. An *instance* is a set of [property](#) values that can be uniquely identified by a [key](#).

388 **3.2.18**389 **key**

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

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

392 **3.2.19**393 **managed object**

394 The actual item in the system environment that is accessed by the [provider](#) — for example, a network
395 interface card.

396 **3.2.20**397 **meta model**

398 A set of [classes](#), [associations](#), and [properties](#) that expresses the types of things that can be defined in a
399 *Schema*. For example, the *meta model* includes a *class* called property that defines the *properties* known
400 to the system, a *class* called method that defines the *methods* known to the system, and a *class* called
401 class that defines the *classes* known to the system.

402 **3.2.21**403 **meta schema**

404 The schema of the meta model.

405 **3.2.22**406 **method**

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

- 409 **3.2.23**
410 **model**
411 A set of [classes](#), [associations](#), and [properties](#) that allows the expression of information about a specific
412 domain. For example, a network may consist of network devices and logical networks. The network
413 devices may have attachment *associations* to each other, and they may have member *associations* to
414 logical networks.
- 415 **3.2.24**
416 **model path**
417 A reference to an object within a namespace.
- 418 **3.2.25**
419 **namespace**
420 An *object* that defines a scope within which object keys must be unique.
- 421 **3.2.26**
422 **namespace path**
423 A reference to a namespace within an implementation that can host CIM objects.
- 424 **3.2.27**
425 **name**
426 The combination of a namespace path and a model path that identifies a unique object.
- 427 **3.2.28**
428 **polymorphism**
429 A [subclass](#) may redefine the implementation of a [method](#) or [property](#) inherited from its [superclass](#). The
430 *property* or *method* is therefore redefined, even if the *superclass* is used to access the object. For
431 example, Device may define availability as a string, and may return the values "powersave," "on," or "off."
432 The Modem *subclass* of Device may redefine (override) availability by returning "on" or "off," but not
433 "powersave". If all Devices are enumerated, any Device that happens to be a modem does not return the
434 value "powersave" for the availability *property*.
- 435 **3.2.29**
436 **property**
437 A value used to characterize an instance of a [class](#). For example, a Device may have a *property* called
438 status.
- 439 **3.2.30**
440 **provider**
441 An executable that can return or set information about a given [managed object](#).
- 442 **3.2.31**
443 **qualifier**
444 A value used to characterize a [method](#), [property](#), or [class](#) in the *meta schema*. For example, if a property
445 has the Key qualifier with the value TRUE, the property is a key for the class.
- 446 **3.2.32**
447 **reference**
448 Special *property types* that are references or pointers to other instances.
- 449 **3.2.33**
450 **schema**
451 A management schema is provided to establish a common conceptual framework at the level of a
452 fundamental topology both for classification and association and for a basic set of classes to establish a

453 common framework to describe the managed environment. A *schema* is a namespace and unit of
454 ownership for a set of classes. *Schemas* may take forms such as a text file, information in a repository, or
455 diagrams in a CASE tool.

456 **3.2.34**

457 **scope**

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

461 **3.2.35**

462 **scoping object**

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

464 **3.2.36**

465 **signature**

466 The return type and parameters supported by a [method](#).

467 **3.2.37**

468 **subclass**

469 See [inheritance](#).

470 **3.2.38**

471 **superclass**

472 See [inheritance](#).

473 **3.2.39**

474 **top-level object 475 (TLO)**

476 A class or object that has no scoping object.

477 **3.2.40**

478 **trigger**

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

482 **4 Symbols and Abbreviated Terms**

483 The following symbols and abbreviations are used in this document.

484 **4.1**

485 **API**

486 application programming interface

487 **4.2**

488 **CIM**

489 Common Information Model

490 **4.3**

491 **DBMS**

492 Database Management System

- 493 **4.4**
494 **DMI**
495 Desktop Management Interface
- 496 **4.5**
497 **GDMO**
498 Guidelines for the Definition of Managed Objects
- 499 **4.6**
500 **HTTP**
501 Hypertext Transfer Protocol
- 502 **4.7**
503 **MIB**
504 Management Information Base
- 505 **4.8**
506 **MIF**
507 Management Information Format
- 508 **4.9**
509 **MOF**
510 Managed Object Format
- 511 **4.10**
512 **OID**
513 object identifier
- 514 **4.11**
515 **SMI**
516 Structure of Management Information
- 517 **4.12**
518 **SNMP**
519 Simple Network Management Protocol
- 520 **4.13**
521 **TLO**
522 top-level object
- 523 **4.14**
524 **UML**
525 Unified Modeling Language

526 **5 Meta Schema**

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

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

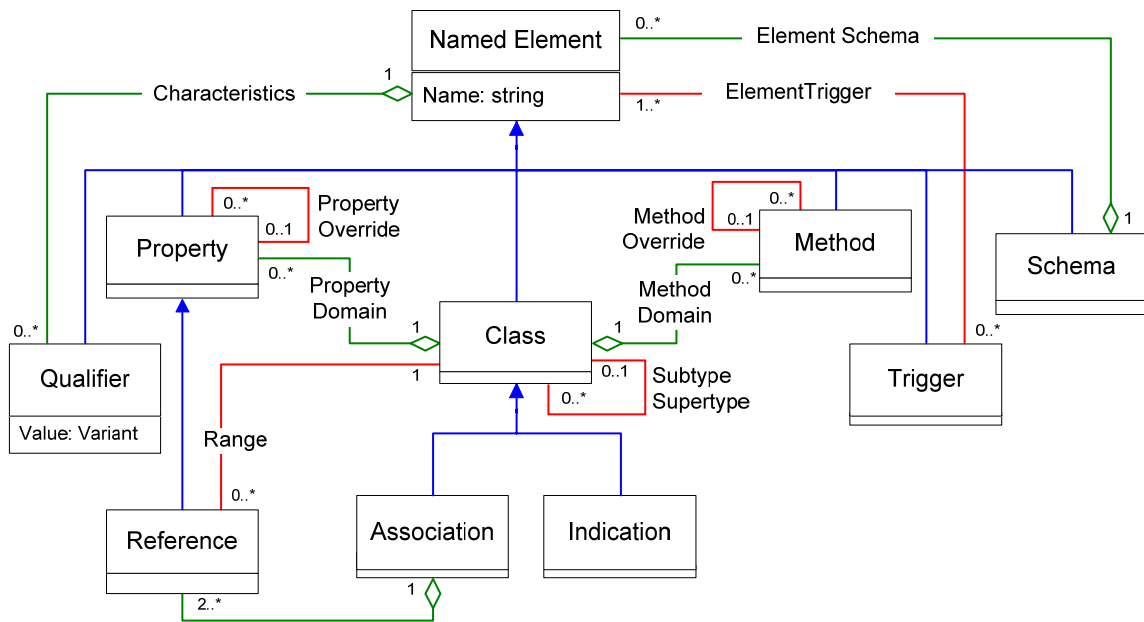
533 5.1 Definition of the Meta Schema

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

- 537 • *Schema*
 538 A group of classes with a single owner. Schemas are used for administration and class naming.
 539 Class names must be unique within their schemas.
- 540 • *Class*
 541 A collection of instances that support the same type (that is, the same properties and methods).
 542 Classes can be arranged in a generalization hierarchy that represents subtype relationships
 543 between classes. The generalization hierarchy is a rooted, directed graph and does not support
 544 multiple inheritance. Classes can have methods, which represent their behavior. A class can
 545 participate in associations as the target of a reference owned by the association. Classes also
 546 have instances (not represented in Figure 2).
- 547 • *Instance*
 548 Each instance provides values for the properties associated with its defining Class. An instance
 549 does not carry values for any other properties or methods not defined in (or inherited by) its
 550 defining class. An instance cannot redefine the properties or methods defined in (or inherited
 551 by) its defining class.
 552 Instances are not named elements and cannot have qualifiers associated with them. However,
 553 qualifiers may be associated with the instance's class, as well as with the properties and
 554 methods defined in or inherited by that class. Instances cannot attach new qualifiers to
 555 properties, methods, or parameters because the association between qualifier and named
 556 element is not restricted to the context of a particular instance.
- 557 • *Property*
 558 Assigns values to characterize instances of a class. A property can be thought of as a pair of
 559 Get and Set functions that return state and set state, respectively, when they are applied to an
 560 object.²
- 561 • *Method*
 562 A declaration of a signature (that is, the method name, return type, and parameters). For a
 563 concrete class, it may imply an implementation.
 564 Properties and methods have reflexive associations that represent property and method
 565 overriding. A method can override an inherited method so that any access to the inherited
 566 method invokes the implementation of the overriding method. Properties are overridden in the
 567 same way.
- 568 • *Trigger*
 569 Recognition of a state change (such as create, delete, update, or access) of a class instance,
 570 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.

- 571 • *Indication*
572 An object created as a result of a trigger. Because indications are subtypes of a class, they can
573 have properties and methods and they can be arranged in a type hierarchy.
- 574 • *Association*
575 A class that contains two or more references. An association represents a relationship between
576 two or more objects. A relationship can be established between classes without affecting any
577 related classes. That is, an added association does not affect the interface of the related
578 classes. Associations have no other significance. Only associations can have references. An
579 association cannot be a subclass of a non-association class. Any subclass of an association is
580 an association.
- 581 • *Reference*
582 Defines the role each object plays in an association. The reference represents the role name of
583 a class in the context of an association. A given object can have multiple relationship instances.
584 For example, a system can be related to many system components.
- 585 • *Qualifier*
586 Characterizes named elements. For example, qualifiers can define the characteristics of a
587 property or the key of a class. Specifically, qualifiers can characterize classes (including
588 associations and indications), properties (including references), methods, and method
589 parameters. Qualifiers do not characterize qualifier types and do not characterize other
590 qualifiers. Qualifiers make the meta schema extensible in a limited and controlled fashion. New
591 types of qualifiers can be added by introducing a new qualifier name, thereby providing new
592 types of meta data to processes that manage and manipulate classes, properties, and other
593 elements of the meta schema.
- 594 Figure 2 provides an overview of the structure of the meta schema. The complete meta schema is
595 defined by the MOF in ANNEX B. The rules defining the meta schema are as follows:
- 596 1) Every meta construct is expressed as a descendent of a named element.
 - 597 2) A named element has zero or more characteristics. A characteristic is a qualifier for a named
598 element.
 - 599 3) A named element can trigger zero or more indications.
 - 600 4) A schema is a named element and can contain zero or more classes. A class must belong to
601 only one schema.
 - 602 5) A qualifier type (not shown in Figure 2) is a named element and must supply a type for a
603 qualifier (that is, a qualifier must have a qualifier type). A qualifier type can be used to type zero
604 or more qualifiers.



605

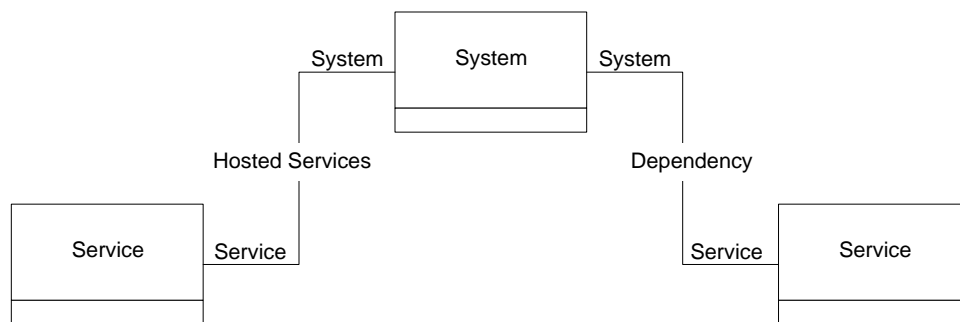
606

Figure 2 – Meta Schema Structure

- 607 6) A qualifier is a named element and has a name, a type (intrinsic data type), a value of this type,
 608 a scope, a flavor, and a default value. The type of the qualifier value must agree with the type of
 609 the qualifier type.
- 610 7) A property is a named element with exactly one domain: the class that owns the property. The
 611 property can apply to instances of the domain (including instances of subclasses of the domain)
 612 and not to any other instances.
- 613 8) A property can override another property from a different class. The domain of the overridden
 614 property must be a supertype of the domain of the overriding property. For non-reference
 615 properties, the type of the overriding property shall be the same as the type of the overridden
 616 property. For References, the range of the overriding Reference shall be the same as, or a
 617 subclass of, the range of the overridden Reference.
- 618 9) The class referenced by the range association (Figure 5) of an overriding reference must be the
 619 same as, or a subtype of, the class referenced by the range associations of the overridden
 620 reference.
- 621 10) The domain of a reference must be an association.
- 622 11) A class is a type of named element. A class can have instances (not shown on the diagram)
 623 and is the domain for zero or more properties. A class is the domain for zero or more methods.
- 624 12) A class can have zero or one supertype and zero or more subtypes.
- 625 13) An association is a type of class. Associations are classes with an association qualifier.
- 626 14) An association must have two or more references.
- 627 15) An association cannot inherit from a non-association class.
- 628 16) Any subclass of an association is an association.
- 629 17) A method is a named element with exactly one domain: the class that owns the method. The
 630 method can apply to instances of the domain (including instances of subclasses of the domain)
 631 and not to any other instances.

- 632 18) A method can override another method from a different class. The domain of the overridden
633 method must be a superclass of the domain of the overriding method.
- 634 19) A trigger is an operation that is invoked on any state change, such as object creation, deletion,
635 modification, or access, or on property modification or access. Qualifiers, qualifier types, and
636 schemas may not have triggers. The changes that invoke a trigger are specified as a qualifier.
- 637 20) An indication is a type of class and has an association with zero or more named triggers that
638 can create instances of the indication.
- 639 21) Every meta-schema object is a descendent of a named element. All names are case-
640 insensitive. The naming rules, which vary depending on the creation type of the object, are as
641 follows:
- 642 a) Fully-qualified class names (that is, prefixed by the schema name) are unique within the
643 schema.
 - 644 b) Fully-qualified association and indication names are unique within the schema (implied by
645 the fact that associations and indications are subtypes of class).
 - 646 c) Implicitly-defined qualifier names are unique within the scope of the characterized object.
647 That is, a named element may not have two characteristics with the same name. Explicitly-
648 defined qualifier names are unique within the defining namespace and must agree in type,
649 scope, and flavor with any explicitly-defined qualifier of the same name.
 - 650 d) Trigger names must be unique within the property, class, or method to which they apply.
 - 651 e) Method and property names must be unique within the domain class. A class can inherit
652 more than one property or method with the same name. Property and method names can
653 be qualified using the name of the declaring class.
 - 654 f) Reference names must be unique within the scope of their defining association and obey
655 the same rules as property names. Reference names do not have to be unique within the
656 scope of the related class because the reference provides the name of the class in the
657 context defined by the association (Figure 3).

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



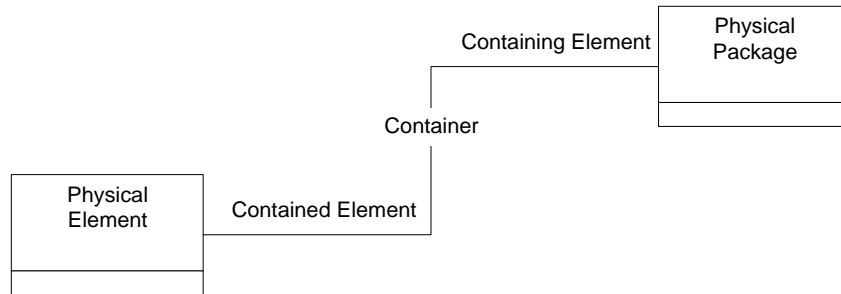
662

663

Figure 3 – Reference Naming

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

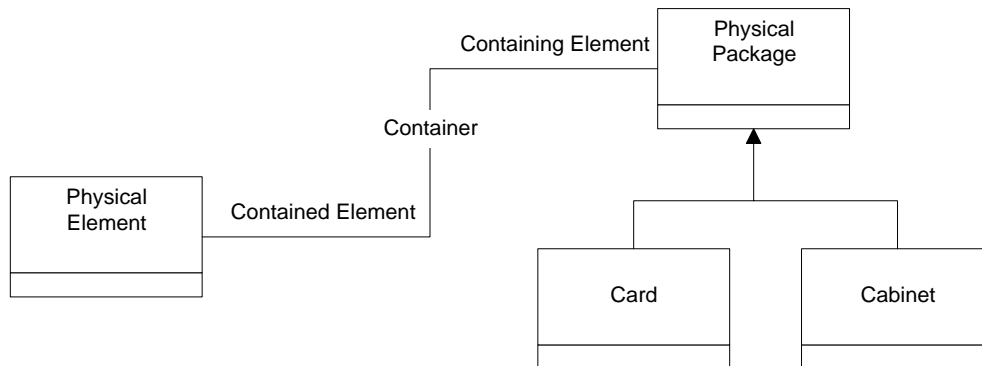
- 669 23) Association and indication are types of class, so they can be the domain for methods,
 670 properties, and references. That is, associations and indications can have properties and
 671 methods just as a class does. Associations and indications can have instances. The instance of
 672 an association has a set of references that relate one or more objects. An instance of an
 673 indication represents an event and is created because of that event — usually a trigger.
 674 Indications are not required to have keys. Typically, indications are very short-lived objects to
 675 communicate information to an event consumer.
- 676 24) A reference has a range that represents the type of the Reference. For example, in the model of
 677 PhysicalElements and PhysicalPackages (Figure 4), there are two references:
 678 – ContainedElement has PhysicalElement as its range and container as its domain.
 679 – ContainingElement has PhysicalPackage as its range and container as its domain.



680

681 **Figure 4 – References, Ranges, and Domains**

- 682 25) A class has a subtype-supertype association for substitutions so that any instance of a subtype
 683 can be substituted for any instance of the supertype in an expression without invalidating the
 684 expression.
- 685 In the container example (Figure 5), Card is a subtype of PhysicalPackage. Therefore, Card can
 686 be used as a value for the ContainingElement reference. That is, an instance of Card can be
 687 used as a substitute for an instance of PhysicalPackage.



688

689 **Figure 5 – References, Ranges, Domains, and Inheritance**

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

- 695 26) The override relationship is used to indicate the substitution relationship between a property or
 696 method of a subclass and the overridden property or method inherited from the superclass. This
 697 is the opposite of the C++ convention in which the superclass property or method is specified as
 698 virtual, with overrides as a side effect of declaring a feature with the same signature as the
 699 inherited virtual feature.
- 700 27) The number of references in an association class defines the arity of the association. An
 701 association containing two references is a binary association. An association containing three
 702 references is a ternary Association. Unary associations, which contain one reference, are not
 703 meaningful. Arrays of references are not allowed. When an association is subclassed, its arity
 704 cannot change.
- 705 28) Schemas allow ownership of portions of the overall model by individuals and organizations who
 706 manage the evolution of the schema. In any given installation, all classes are visible, regardless
 707 of schema ownership. Schemas have a universally unique name. The schema name is part of
 708 the class name. The full class name (that is, class name plus owning schema name) is unique
 709 within the namespace and is the fully-qualified name (see 5.4).

710 5.2 Data Types

711 Properties, references, parameters, and methods (that is, method return values) have a data type. These
 712 data types are limited to the intrinsic data types or arrays of such. Additional constraints apply to the data
 713 types of some elements, as defined in this document. Structured types are constructed by designing new
 714 classes. There are no subtype relationships among the intrinsic data types uint8, sint8, uint16, sint16,
 715 uint32, sint32, uint64, sint64, string, boolean, real32, real64, datetime, char16, and arrays of them. CIM
 716 elements of any intrinsic data type (including <classname> REF) may have the special value NULL,
 717 indicating absence of value, unless further constrained in this document.

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

719

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

720 5.2.1 Datetime Type

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

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

725 `yyyymmddhhmmss.mmmmmmsutc`

726 The meaning of each field is as follows:

- 727 • `yyyy` is a 4-digit year.
- 728 • `mm` is the month within the year (starting with 01).
- 729 • `dd` is the day within the month (starting with 01).
- 730 • `hh` is the hour within the day (24-hour clock, starting with 00).
- 731 • `mm` is the minute within the hour (starting with 00).
- 732 • `ss` is the second within the minute (starting with 00).
- 733 • `mmmmmm` is the microsecond within the second (starting with 000000).
- 734 • `s` is a + (plus) or – (minus), indicating that the value is a timestamp with the sign of Universal
735 Coordinated Time (UTC), which is basically the same as Greenwich Mean Time correction field.
736 A + (plus) is used for time zones east of Greenwich, and a – (minus) is used for time zones
737 west of Greenwich.
- 738 • `utc` is the offset from UTC in minutes (using the sign indicated by `s`).

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

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

744 For example, Monday, May 25, 1998, at 1:30:15 PM EST is represented as 19980525133015.0000000-
745 300.

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

747 The format for intervals is as follows:

748 `dddddddhhmmss.mmmmmm:000`, with

749 The meaning of each field is as follows:

- 750 • `ddddddd` is the number of days.
- 751 • `hh` is the remaining number of hours.
- 752 • `mm` is the remaining number of minutes.
- 753 • `ss` is the remaining number of seconds.
- 754 • `mmmmmm` is the remaining number of microseconds.
- 755 • `:` (colon) indicates that the value is an interval.
- 756 • `000` (the UTC offset field) is always zero for interval properties.

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

759 00000001132312.000000:000.

760 For both timestamps and intervals, the field values shall be zero-padded so that the entire string is always
761 25 characters in length.

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

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

770 The following operations are defined on datetime types:

771 • Arithmetic operations:

- 772 – Adding or subtracting an interval to or from an interval results in an interval.
- 773 – Adding or subtracting an interval to or from a timestamp results in a timestamp.
- 774 – Subtracting a timestamp from a timestamp results in an interval.
- 775 – Multiplying an interval by a numeric or vice versa results in an interval.
- 776 – Dividing an interval by a numeric results in an interval.

777 Other arithmetic operations are not defined.

778 • Comparison operations:

- 779 – Testing for equality of two timestamps or two intervals results in a Boolean value.
- 780 – Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in
781 a Boolean value.

782 Other comparison operations are not defined.

783 Comparison between a timestamp and an interval and vice versa is not defined.

784 Specifications that use the definition of these operations (such as specifications for query languages)
785 should state how undefined operations are handled.

786 Any operations on datetime types in an expression shall be handled as if the following sequential steps
787 were performed:

788 1) Each datetime value is converted into a range of microsecond values, as follows:

- 789 • The lower bound of the range is calculated from the datetime value, with any asterisks
790 replaced by their minimum value.
- 791 • The upper bound of the range is calculated from the datetime value, with any asterisks
792 replaced by their maximum value.
- 793 • The basis value for timestamps is the oldest valid value (that is, 0 microseconds
794 corresponds to 00:00.000000 in the timezone with datetime offset +720, on January 1 in
795 the year 1 BCE, using the proleptic Gregorian calendar). This definition implicitly performs
796 timestamp normalization. Note that 1 BCE is the year before 1 CE.

- 797 2) The expression is evaluated using the following rules for any datetime ranges:
 798
- 799 • Definitions:
 800 $T(x, y)$ The microsecond range for a timestamp with the lower bound x and the upper bound y
 801 $I(x, y)$ The microsecond range for an interval with the lower bound x and the upper bound y
 802
 803 $D(x, y)$ The microsecond range for a datetime (timestamp or interval) with the lower bound x and the upper bound y
 804
 - 805 • Rules:
 806 $I(a, b) + I(c, d) := I(a+c, b+d)$
 807 $I(a, b) - I(c, d) := I(a-d, b-c)$
 808 $T(a, b) + I(c, d) := T(a+c, b+d)$
 809 $T(a, b) - I(c, d) := T(a-d, b-c)$
 810 $T(a, b) - T(c, d) := I(a-d, b-c)$
 811 $I(a, b) * c := I(a*c, b*c)$
 812 $I(a, b) / c := I(a/c, b/c)$
 813 $D(a, b) < D(c, d) :=$ true if $b < c$, false if $a \geq d$, otherwise NULL (uncertain)
 814 $D(a, b) \leq D(c, d) :=$ true if $b \leq c$, false if $a > d$, otherwise NULL (uncertain)
 815 $D(a, b) > D(c, d) :=$ true if $a > d$, false if $b \leq c$, otherwise NULL (uncertain)
 816 $D(a, b) \geq D(c, d) :=$ true if $a \geq d$, false if $b < c$, otherwise NULL (uncertain)
 817 $D(a, b) = D(c, d) :=$ true if $a = b = c = d$, false if $b < c$ OR $a > d$, otherwise NULL
 818 (uncertain)
 819 $D(a, b) \neq D(c, d) :=$ true if $b < c$ OR $a > d$, false if $a = b = c = d$, otherwise NULL
 820 (uncertain)
 821 These rules follow the well-known mathematical interval arithmetic. For a definition of
 822 mathematical interval arithmetic, see http://en.wikipedia.org/wiki/Interval_arithmetic.
 823 NOTE 1: Mathematical interval arithmetic is commutative and associative for addition and
 824 multiplication, as in ordinary arithmetic.
 825 NOTE 2: Mathematical interval arithmetic mandates the use of three-state logic for the result of
 826 comparison operations. A special value called "uncertain" indicates that a decision cannot be made.
 827 The special value of "uncertain" is mapped to the NULL value in datetime comparison operations.
- 828 3) Overflow and underflow condition checking is performed on the result of the expression, as
 829 follows:
- 830 For timestamp results:
- 831 • A timestamp older than the oldest valid value in the timezone of the result produces
 832 an arithmetic underflow condition.
 - 833 • A timestamp newer than the newest valid value in the timezone of the result produces
 834 an arithmetic overflow condition.
- 835 For interval results:
- 836 • A negative interval produces an arithmetic underflow condition.
 - 837 • A positive interval greater than the largest valid value produces an arithmetic overflow
 838 condition.

839 Specifications using these operations (for instance, query languages) should define how these
840 conditions are handled.

841 4) If the result of the expression is a datetime type, the microsecond range is converted into a valid
842 datetime value such that the set of asterisks (if any) determines a range that matches the actual
843 result range or encloses it as closely as possible. The GMT timezone shall be used for any
844 timestamp results.

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

846 EXAMPLE:

847 "20051003110000.000000+000" + "00000000002233.000000:000" evaluates to
848 "20051003112233.000000+000"

849 "20051003110000.*****+000" + "00000000002233.000000:000" evaluates to
850 "20051003112233.*****+000"

851 "20051003110000.*****+000" + "00000000002233.00000*:000" evaluates to
852 "200510031122**.******+000"

853 "20051003110000.*****+000" + "00000000002233.*****:000" evaluates to
854 "200510031122**.******+000"

855 "20051003110000.*****+000" + "00000000005959.*****:000" evaluates to
856 "20051003*****.******+000"

857 "20051003110000.*****+000" + "000000000022**.******:000" evaluates to
858 "2005100311****.******+000"

859 "20051003112233.000000+000" - "00000000002233.000000:000" evaluates to
860 "20051003110000.000000+000"

861 "20051003112233.*****+000" - "00000000002233.000000:000" evaluates to
862 "20051003110000.*****+000"

863 "20051003112233.*****+000" - "00000000002233.00000*:000" evaluates to
864 "20051003110000.*****+000"

865 "20051003112233.*****+000" - "00000000002232.*****:000" evaluates to
866 "200510031100**.******+000"

867 "20051003112233.*****+000" - "00000000002233.*****:000" evaluates to
868 "20051003*****.******+000"

869 "20051003060000.000000-300" + "00000000002233.000000:000" evaluates to
870 "20051003112233.000000+000"

871 "20051003060000.*****-300" + "00000000002233.000000:000" evaluates to
872 "20051003112233.*****+000"

873 "000000000011**.******:000" * 60 evaluates to
874 "0000000011****.******:000"

875 60 times adding up "000000000011**.******:000" evaluates to
876 "0000000011****.******:000"

877 "20051003112233.000000+000" = "20051003112233.000000+000" evaluates to true
878 "20051003122233.000000+060" = "20051003112233.000000+000" evaluates to true
879 "20051003112233.*****+000" = "20051003112233.*****+000" evaluates to NULL (uncertain)
880 "20051003112233.*****+000" = "200510031122**.******+000" evaluates to NULL (uncertain)
881 "20051003112233.*****+000" = "20051003112234.*****+000" evaluates to false
882 "20051003112233.*****+000" < "20051003112234.*****+000" evaluates to true
883 "20051003112233.5*****+000" < "20051003112233.*****+000" evaluates to NULL (uncertain)

884 A datetime value is valid if the value of each single field is in the valid range. Valid values shall
885 not be rejected by any validity checking within the CIM infrastructure.

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

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

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

891 • For timestamp values:

892	Oldest valid timestamp	"00000101000000.000000+720"
893		Reserved range (1 million values)
894	Oldest useable timestamp	"00000101000001.000000+720"
895		Range interpreted as points in time
896	Youngest useable timestamp	"99991231115959.999998-720"
897		Reserved range (1 value)
898	Youngest valid timestamp	"99991231115959.999999-720"

899 – Special values in the reserved ranges:

900	"Now"	"00000101000000.000000+720"
901	"Infinite past"	"00000101000000.999999+720"
902	"Infinite future"	"99991231115959.999999-720"

903 • For interval values:

904	Smallest valid and useable interval	"00000000000000.000000:000"
905		Range interpreted as durations
906	Largest useable interval	"99999999235958.999999:000"
907		Reserved range (1 million values)
908	Largest valid interval	"99999999235959.999999:000"

909 – Special values in reserved range:

910	"Infinite duration"	"99999999235959.000000:000"
-----	---------------------	-----------------------------

911 5.2.2 Indicating Additional Type Semantics with Qualifiers

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

```

916     class Acme_Example
917     {
918         [counter]
919         uint32 NumberOfCycles;
920         [gauge]
921         uint32 MaxTemperature;
922         [octetstring, ArrayType("Indexed")]
923         uint8 IPAddress[10];
924     };
  
```

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

927 5.3 Supported Schema Modifications

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

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

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

951 5.3.1 Schema Versions

952 Schema versioning is described in the [DSP4004](#). Versioning takes the form m.n.u, where:

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

956 The usage rules for the Version qualifier in 5.5.2.53 provide additional information.

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

965 Table 3 lists modifications to the CIM schemas in final status that cause a major version number change.
 966 Preliminary models are allowed to evolve based on implementation experience. These modifications
 967 change application behavior and/or customer code. Therefore, they force a major version update and are
 968 discouraged. Table 3 is an exhaustive list of the possible modifications based on current CIM experience
 969 and knowledge. Items could be added as new issues are raised and CIM standards evolve.

970 Alterations beyond those listed in Table 3 are considered interface-preserving and require the minor
 971 version number to be incremented. Updates/errata are not classified as major or minor in their impact, but
 972 they are required to correct errors or to coordinate across standards bodies.

973 **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	

974 **5.4 Class Names**

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

978 The format of the fully-qualified name allows the scope of class names to be limited to a schema. That is,
 979 the schema name is assumed to be unique, and the class name is required to be unique only within the

980 schema. The isolation of the schema name using the underscore character allows user interfaces
981 conveniently to strip off the schema when the schema is implied by the context.

982 The following are examples of fully-qualified class names:

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

987 5.5 Qualifiers

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

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

999 5.5.1 Meta Qualifiers

1000 Table 4 lists the qualifiers that refine the definition of the meta constructs in the model. These qualifiers
1001 refine the actual usage of a class declaration and are mutually exclusive.

1002 **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.

1003 5.5.2 Standard Qualifiers

1004 The following subclauses list the standard qualifiers required for all CIM-compliant implementations. Any
1005 given object does not have all the qualifiers listed. Additional qualifiers can be supplied by extension
1006 classes to provide instances of the class and other operations on the class.

1007 Not all of these qualifiers can be used together. The following principles apply:

- 1008 • Not all qualifiers can be applied to all meta-model constructs. For each qualifier, the constructs to
1009 which it applies are listed.
- 1010 • For a particular meta-model construct, such as associations, the use of the legal qualifiers may be
1011 further constrained because some qualifiers are mutually exclusive or the use of one qualifier implies
1012 restrictions on the value of another, and so on. These usage rules are documented in the subclause
1013 for each qualifier.
- 1014 • Legal qualifiers are not inherited by meta-model constructs. For example, the MaxLen qualifier that
1015 applies to properties is not inherited by references.

1016 The meta-model constructs that can use a particular qualifier are identified for each qualifier. For
1017 qualifiers such as Association (see 5.5.1), there is an implied usage rule that the meta qualifier must also
1018 be present. For example, the implicit usage rule for the Aggregation qualifier (see 5.5.2.3) is that the
1019 Association qualifier must also be present.

1020 The allowed set of values for scope is (Class Association Indication Property Reference Parameter
1021 Method). Each qualifier has one or more of these scopes. If the scope is Class it does not apply to
1022 Association or Indication. If the scope is Property it does not apply to Reference.

1023 **5.5.2.1 Abstract**

1024 The Abstract qualifier takes Boolean values, and has a Scope(Class Association Indication). The default
1025 value is FALSE.

1026 This qualifier indicates that the class is abstract and serves only as a base for new classes. It is not
1027 possible to create instances of such classes.

1028 **5.5.2.2 Aggregate**

1029 The Aggregate qualifier takes Boolean values, and has a Scope(Reference). The default value is FALSE.

1030 The Aggregation and Aggregate qualifiers are used together. The Aggregation qualifier relates to the
1031 association, and the Aggregate qualifier specifies the parent reference.

1032 **5.5.2.3 Aggregation**

1033 The Aggregation qualifier takes Boolean values, and has Scope(Association). The default value is
1034 FALSE.

1035 The Aggregation qualifier indicates that the association is an aggregation.

1036 **5.5.2.4 ArrayType**

1037 The ArrayType qualifier takes string array values, and has Scope(Property Parameter). The default value
1038 is FALSE.

1039 The ArrayType qualifier is the type of the qualified array. Valid values are "Bag", "Indexed," and
1040 "Ordered."

1041 For definitions of the array types, refer to 7.8.2.

1042 The ArrayType qualifier shall be applied only to properties and method parameters that are arrays
1043 (defined using the square bracket syntax specified in ANNEX A).

1044 **5.5.2.5 Bitmap**

1045 The Bitmap qualifier takes string array values, and has a Scope(Property Parameter Method). The default
1046 value is NULL.

1047 The Bitmap qualifier indicates the bit positions that are significant in a bitmap. The bitmap is evaluated
1048 from the right, starting with the least significant value. This value is referenced as 0 (zero). For example,
1049 using a uint8 data type, the bits take the form Mxxx xxxL, where M and L designate the most and least
1050 significant bits, respectively. The least significant bits are referenced as 0 (zero), and the most significant
1051 bit is 7. The position of a specific value in the Bitmap array defines an index used to select a string literal
1052 from the BitValues array.

1053 The number of entries in the BitValues and Bitmap arrays shall match.

1054 **5.5.2.6 BitValues**

1055 The BitValues qualifier takes string array values, and has Scope(Property Parameter Method). The
1056 default value is NULL.

1057 The BitValues qualifier translates between a bit position value and an associated string. See 5.5.2.5 for
1058 the description for the Bitmap qualifier.

1059 The number of entries in the BitValues and Bitmap arrays shall match.

1060 **5.5.2.7 ClassConstraint**

1061 The ClassConstraint qualifier takes string array values and has Scope(Class Association Indication). The
1062 default value is NULL.

1063 The qualified element specifies one or more constraints that are defined in the Object Constraint
1064 Language (OCL), as specified in the OMG [Object Constraint Language Specification](#).

1065 The ClassConstraint array contains string values that specify OCL definition and invariant constraints.
1066 The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of the qualified
1067 class, association, or indication.

1068 OCL definition constraints define OCL attributes and OCL operations that are reusable by other OCL
1069 constraints in the same OCL context.

1070 The attributes and operations in the OCL definition constraints shall be visible for:

- 1071 • OCL definition and invariant constraints defined in subsequent entries in the same
1072 ClassConstraint array
- 1073 • OCL constraints defined in PropertyConstraint qualifiers on properties and references in a class
1074 whose value (specified or inherited) of the ClassConstraint qualifier defines the OCL definition
1075 constraint
- 1076 • Constraints defined in MethodConstraint qualifiers on methods defined in a class whose value
1077 (specified or inherited) of the ClassConstraint qualifier defines the OCL definition constraint

1078 A string value specifying an OCL definition constraint shall conform to the following syntax:

1079 `ocl_definition_string = "def" [ocl_name] ":" ocl_statement`

1080 Where:

1081 `ocl_name` is the name of the OCL constraint.

1082 `ocl_statement` is the OCL statement of the definition constraint, which defines the reusable attribute
1083 or operation.

1084 An OCL invariant constraint is expressed as a typed OCL expression that specifies whether the constraint
1085 is satisfied. The type of the expression shall be Boolean. The invariant constraint shall be satisfied at any
1086 time in the lifetime of the instance.

1087 A string value specifying an OCL invariant constraint shall conform to the following syntax:

1088 `ocl_invariant_string = "inv" [ocl_name] ":" ocl_statement`

1089 Where:

1090 `ocl_name` is the name of the OCL constraint.

1091 ocl_statement is the OCL statement of the invariant constraint, which defines the Boolean
1092 expression.

1093 EXAMPLE: For example, to check that both property x and property y cannot be NULL in any instance of a class,
1094 use the following qualifier, defined on the class:

```
1095 ClassConstraint {
1096     "inv: not (self.x.ocIsUndefined() and self.y.ocIsUndefined())"
1097 }
```

1098 The same check can be performed by first defining OCL attributes. Also, the invariant constraint is named
1099 in the following example:

```
1100 ClassConstraint {
1101     "def: xNull : Boolean = self.x.ocIsUndefined()",
1102     "def: yNull : Boolean = self.y.ocIsUndefined()",
1103     "inv xyNullCheck: xNull = false or yNull = false"
1104 }
```

1105 5.5.2.8 Composition

1106 The Composition qualifier takes Boolean values and has Scope(Association). The default value is FALSE.

1107 The Composition qualifier refines the definition of an aggregation association, adding the semantics of a
1108 whole-part/compositional relationship to distinguish it from a collection or basic aggregation. This
1109 refinement is necessary to map CIM associations more precisely into UML where whole-part relationships
1110 are considered compositions. The semantics conveyed by composition align with that of the [OMG UML](#)
1111 [Specification](#). Following is a quote (with emphasis added) from section 7.3.3:

1112 "Composite aggregation is a strong form of aggregation that requires a part instance be included in at
1113 most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it."

1114 Use of this qualifier imposes restrictions on the membership of the 'collecting' object (the whole). Care
1115 should be taken when entities are added to the aggregation, because they shall be "parts" of the whole.
1116 Also, if the collecting entity (the whole) is deleted, it is the responsibility of the implementation to dispose
1117 of the parts. The behavior may vary with the type of collecting entity whether the parts are also deleted.
1118 This is very different from that of a collection, because a collection may be removed without deleting the
1119 entities that are collected.

1120 The Aggregation and Composition qualifiers are used together. Aggregation indicates the general nature
1121 of the association, and Composition indicates more specific semantics of whole-part relationships. This
1122 duplication of information is necessary because Composition is a more recent addition to the list of
1123 qualifiers. Applications can be built only on the basis of the earlier Aggregation qualifier.

1124 5.5.2.9 Correlatable

1125 The Correlatable qualifier takes string array values, and has Scope(Property). The default value is NULL.

1126 The Correlatable qualifier is used to define sets of properties that can be compared to determine if two
1127 CIM instances represent the same resource entity. For example, these instances may cross
1128 logical/physical boundaries, CIM Server scopes, or implementation interfaces.

1129 The sets of properties to be compared are defined by first specifying the organization in whose context
1130 the set exists (organization_name), and then a set name (set_name). In addition, a property is given a
1131 role name (role_name) to allow comparisons across the CIM Schema (that is, where property names may
1132 vary although the semantics are consistent).

1133 The value of each entry in the Correlatable qualifier string array shall follow the formal syntax:

```
1134 correlatablePropertyID = organization_name ":" set_name ":" role_name
```

1135 The determination whether two CIM instances represent the same resource entity is done by comparing
 1136 one or more property values of each instance (where the properties are tagged by their role name), as
 1137 follows: The property values of all role names within at least one matching organization name / set name
 1138 pair shall match in order to conclude that the two instances represent the same resource entity.
 1139 Otherwise, no conclusion can be reached and the instances may or may not represent the same resource
 1140 entity.

1141 `correlatablePropertyID` values shall be compared case-insensitively. For example,
 1142 "Acme:Set1:Role1" and "ACME:set1:role1" are considered matching. Note that the values of any
 1143 string properties in CIM are defined to be compared case-sensitively.

1144 To assure uniqueness of a `correlatablePropertyID`:

- 1145 • `organization_name` shall include a copyrighted, trademarked or otherwise unique name that is
 1146 owned by the business entity defining `set_name`, or is a registered ID that is assigned to the
 1147 business entity by a recognized global authority. `organization_name` shall not contain a colon
 1148 (":"). For DMTF defined `correlatablePropertyID` values, the `organization_name` shall be
 1149 "CIM".
- 1150 • `set_name` shall be unique within the context of `organization_name` and identifies a specific set
 1151 of correlatable properties. `set_name` shall not contain a colon (":").
- 1152 • `role_name` shall be unique within the context of `organization_name` and `set_name` and identifies
 1153 the semantics or role that the property plays within the Correlatable comparison.

1154 The Correlatable qualifier may be defined on only a single class. In this case, instances of only that class
 1155 are compared. However, if the same correlation set (defined by `organization_name` and `set_name`) is
 1156 specified on multiple classes, then comparisons can be done across those classes.

1157 EXAMPLE: As an example, assume that instances of two classes can be compared: Class1 with properties
 1158 PropA, PropB, and PropC, and Class2 with properties PropX, PropY and PropZ. There are two correlation sets
 1159 defined, one set with two properties that have the role names Role1 and Role2, and the other set with one property
 1160 with the role name OnlyRole. The following MOF represents this example:

```

1161 Class1 {
1162     [Correlatable {"Acme:Set1:Role1"}]
1163     string PropA;
1164     [Correlatable {"Acme:Set2:OnlyRole"}]
1165     string PropB;
1166     [Correlatable {"Acme:Set1:Role2"}]
1167     string PropC;
1168 };
1169 Class2 {
1170     [Correlatable {"Acme:Set1:Role1"}]
1171     string PropX;
1172     [Correlatable {"Acme:Set2:OnlyRole"}]
1173     string PropY;
1174     [Correlatable {"Acme:Set1:Role2"}]
1175     string PropZ;
1176 };
  
```

1177 Following the comparison rules defined above, one can conclude that an instance of Class1 and an
 1178 instance of Class2 represent the same resource entity if PropB and PropY's values match, or if
 1179 PropA/PropX and PropC/PropZ's values match, respectively.

1180 The Correlatable qualifier can be used to determine if multiple CIM instances represent the same
 1181 underlying resource entity. Some may wonder if an instance's key value (such as InstanceID) is meant to
 1182 perform the same role. This is not the case. InstanceID is merely an opaque identifier of a CIM instance,

1183 whereas Correlatable is not opaque and can be used to draw conclusions about the identity of the
1184 underlying resource entity of two or more instances.

1185 DMTF-defined Correlatable qualifiers are defined in the CIM Schema on a case-by-case basis. There is
1186 no central document that defines them.

1187 **5.5.2.10 Counter**

1188 The Counter qualifier takes string array values and has Scope(Property Parameter Method). The default
1189 value is FALSE.

1190 The Counter qualifier applies only to unsigned integer types.

1191 It represents a non-negative integer that monotonically increases until it reaches a maximum value of
1192 $2^N - 1$, when it wraps around and starts increasing again from zero. N can be 8, 16, 32, or 64 depending
1193 on the data type of the object to which the qualifier is applied. Counters have no defined initial value, so a
1194 single value of a counter generally has no information content.

1195 **5.5.2.11 Deprecated**

1196 The Deprecated qualifier takes string array values and has Scope(Class Association Indication Property
1197 Reference Parameter Method). The default value is NULL.

1198 The Deprecated qualifier indicates that the CIM element (for example, a class or property) that the
1199 qualifier is applied to is considered deprecated. The qualifier may specify replacement elements. Existing
1200 instrumentation shall continue to support the deprecated element so that current applications do not
1201 break. Existing instrumentation should add support for any replacement elements. A deprecated element
1202 should not be used in new applications. Existing and new applications shall tolerate the deprecated
1203 element and should move to any replacement elements as soon as possible. The deprecated element
1204 may be removed in a future major version release of the CIM schema, such as CIM 2.x to CIM 3.0.

1205 The qualifier acts inclusively. Therefore, if a class is deprecated, all the properties, references, and
1206 methods in that class are also considered deprecated. However, no subclasses or associations or
1207 methods that reference that class are deprecated unless they are explicitly qualified as such. For clarity
1208 and to specify replacement elements, all such implicitly deprecated elements should be specifically
1209 qualified as deprecated.

1210 The Deprecated qualifier's string value should specify one or more replacement elements. Replacement
1211 elements shall be specified using the following syntax:

```
1212   className [ [ embeddedInstancePath ] "." elementSpec ];
```

1213 where:

```
1214   elementSpec = propertyName | methodName "(" [ parameterName *(", " parameterName) ] ")"
```

1215 is a specification of the replacement element.

```
1216   embeddedInstancePath = 1*( "." propertyName )
```

1217 is a specification of a path through embedded instances.

1218 The qualifier is defined as a string array so that a single element can be replaced by multiple elements.

1219 If there is no replacement element, then the qualifier string array shall contain a single entry with the
1220 string "No value".

- 1221 When an element is deprecated, its description shall indicate why it is deprecated and how any
1222 replacement elements are used. Following is an acceptable example description:
- 1223 "The X property is deprecated in lieu of the Y method defined in this class because the property
1224 actually causes a change of state and requires an input parameter."
- 1225 The parameters of the replacement method may be omitted.
- 1226 NOTE 1: Replacing a deprecated element with a new element results in duplicate representations of the element.
1227 This is of particular concern when deprecated classes are replaced by new classes and instances may be duplicated.
1228 To allow a management application to detect such duplication, implementations should document (in a ReadMe,
1229 MOF, or other documentation) how such duplicate instances are detected.
- 1230 NOTE 2: Key properties may be deprecated, but they shall continue to be key properties and shall satisfy all rules for
1231 key properties. When a key property is no longer intended to be a key, only one option is available. It is necessary to
1232 deprecate the entire class and therefore its properties, methods, references, and so on, and to define a new class
1233 with the changed key structure.
- 1234 **5.5.2.12 Description**
- 1235 The Description qualifier takes string array values, and has a Scope(Class Association Indication Property
1236 Reference Parameter Method). The default value is NULL.
- 1237 The Description qualifier describes a named element.
- 1238 **5.5.2.13 DisplayName**
- 1239 The DisplayName qualifier takes string values and has Scope(Class Association Indication Property
1240 Reference Parameter Method). The default value is NULL.
- 1241 The DisplayName qualifier defines a name that is displayed on a user interface instead of the actual
1242 name of the element.
- 1243 **5.5.2.14 DN**
- 1244 The DN qualifier takes string array values, and has a Scope(Property Parameter Method). The default
1245 value is FALSE.
- 1246 When applied to a string element, the DN qualifier specifies that the string shall be a distinguished name
1247 as defined in Section 9 of [X.501](#) and the string representation defined in [RFC2253](#). This qualifier shall not
1248 be applied to qualifiers that are not of the intrinsic data type string.
- 1249 **5.5.2.15 EmbeddedInstance**
- 1250 The EmbeddedInstance qualifier takes string array values and has Scope(Property Parameter Method).
1251 The default value is NULL.
- 1252 The qualified string typed element contains an embedded instance. The encoding of the instance
1253 contained in the string typed element qualified by EmbeddedInstance follows the rules defined in
1254 ANNEX G.
- 1255 This qualifier may be used only on elements of string type.
- 1256 The qualifier value shall specify the name of a CIM class in the same namespace as the class owning the
1257 qualified element. The embedded instance shall be an instance of the specified class, including instances
1258 of its subclasses.
- 1259 This qualifier shall not be used on an element that overrides an element not qualified by
1260 EmbeddedInstance. However, it may be used on an overriding element to narrow the class specified in
1261 this qualifier on the overridden element to one of its subclasses.

1262 See ANNEX G for examples.

1263 **5.5.2.16 EmbeddedObject**

1264 The EmbeddedObject qualifier takes Boolean values and has Scope(Property Parameter Method). The
1265 default value is FALSE.

1266 This qualifier indicates that the qualified string typed element contains an encoding of an instance's data
1267 or an encoding of a class definition. The encoding of the object contained in the string typed element
1268 qualified by EmbeddedObject follows the rules defined in ANNEX G.

1269 This qualifier may be used only on elements of string type. It shall not be used on an element that
1270 overrides an element not qualified by EmbeddedObject.

1271 See ANNEX G for examples.

1272 **5.5.2.17 Exception**

1273 The Exception qualifier takes Boolean values and has Scope(Class Indication). The default value is
1274 FALSE.

1275 This qualifier indicates that the class and all subclasses of this class describe transient exception
1276 information. The definition of this qualifier is identical to that of the Abstract qualifier except that it cannot
1277 be overridden. It is not possible to create instances of exception classes.

1278 The Exception qualifier denotes a class hierarchy that defines transient (very short-lived) exception
1279 objects. Instances of Exception classes communicate exception information between CIMEntities. The
1280 Exception qualifier cannot be used with the Abstract qualifier. The subclass of an exception class shall be
1281 an exception class.

1282 **5.5.2.18 Experimental**

1283 The Experimental qualifier takes Boolean values and has Scope(Class Association Indication Property
1284 Reference Parameter Method). The default value is FALSE.

1285 If the Experimental qualifier is specified, the qualified element has experimental status. The implications
1286 of experimental status are specified by the schema owner.

1287 In a DMTF-produced schema, experimental elements are subject to change and are not part of the final
1288 schema. In particular, the requirement to maintain backwards compatibility across minor schema versions
1289 does not apply to experimental elements. Experimental elements are published for developing
1290 implementation experience. Based on implementation experience, changes may occur to this element in
1291 future releases, it may be standardized "as is," or it may be removed. An implementation does not have to
1292 support an experimental feature to be compliant to a DMTF-published schema.

1293 When applied to a class, the Experimental qualifier conveys experimental status to the class itself, as well
1294 as to all properties and features defined on that class. Therefore, if a class already bears the
1295 Experimental qualifier, it is unnecessary also to apply the Experimental qualifier to any of its properties or
1296 features, and such redundant use is discouraged.

1297 No element shall be both experimental and deprecated (as with the Deprecated qualifier). Experimental
1298 elements whose use is considered undesirable should simply be removed from the schema.

1299 **5.5.2.19 Gauge**

1300 The Gauge qualifier takes Boolean values and has Scope(Property Parameter Method). The default value
1301 is FALSE.

1302 The Gauge qualifier is applicable only to unsigned integer types. It represents an integer that may
1303 increase or decrease in any order of magnitude.

1304 The value of a gauge is capped at the implied limits of the property's data type. If the information being
1305 modeled exceeds an implied limit, the value represented is that limit. Values do not wrap. For unsigned
1306 integers, the limits are zero (0) to 2^n-1 , inclusive. For signed integers, the limits are $-(2^{n-1})$ to
1307 $2^{n-1}-1$, inclusive. N can be 8, 16, 32, or 64 depending on the data type of the property to which the
1308 qualifier is applied.

1309 **5.5.2.20 IN**

1310 The IN qualifier takes Boolean values and has Scope(Parameter). The default value is TRUE.

1311 The IN qualifier is used with an associated parameter to pass values to a method.

1312 **5.5.2.21 IsPUnit**

1313 The IsPUnit qualifier takes Boolean values and has Scope(Property Parameter Method). The default
1314 value is FALSE.

1315 The qualified string typed property, method return value, or method parameter represents a programmatic
1316 unit of measure. The value of the string element follows the syntax for programmatic units.

1317 The qualifier must be used on string data types only. A value of NULL for the string element indicates that
1318 the programmatic unit is unknown. The syntax for programmatic units is defined in ANNEX C.

1319 Experimental: This qualifier has status "Experimental."

1320 **5.5.2.22 Key**

1321 The Key qualifier takes Boolean values and has Scope(Property Reference). The default value is FALSE.

1322 The property or reference is part of the model path (see 8.3.2 for information on the model path). If more
1323 than one property or reference has the Key qualifier, then all such elements collectively form the key (a
1324 compound key).

1325 The values of key properties and key references are determined once at instance creation time and shall
1326 not be modified afterwards. Properties of an array type shall not be qualified with Key. Properties qualified
1327 with EmbeddedObject or EmbeddedInstance shall not be qualified with Key. Key properties and Key
1328 references shall not be NULL.

1329 **5.5.2.23 MappingStrings**

1330 The MappingStrings qualifier takes string array values and has Scope(Class Association Indication
1331 Property Reference Parameter Method). The default value is NULL.

1332 This qualifier indicates mapping strings for one or more management data providers or agents. See 5.5.5
1333 for details.

1334 **5.5.2.24 Max**

1335 The Max qualifier takes uint32 values and has Scope(Reference). The default value is NULL.

1336 The Max qualifier specifies the maximum cardinality of the reference, which is the maximum number of
1337 values a given reference may have for each set of other reference values in the association. For example,
1338 if an association relates A instances to B instances, and there shall be at most one A instance for each B
1339 instance, then the reference to A should have a Max(1) qualifier.

1340 The NULL value means that the maximum cardinality is unlimited.

1341 **5.5.2.25 MaxLen**

1342 The MaxLen qualifier takes uint32 values and has Scope(Property Parameter Method). The default value
1343 is NULL.

1344 The MaxLen qualifier specifies the maximum length, in characters, of a string data item. MaxLen may be
1345 used only on string data types. If MaxLen is applied to CIM elements with a string array data type, it
1346 applies to every element of the array. A value of NULL implies unlimited length.

1347 An overriding property that specifies the MAXLEN qualifier must specify a maximum length no greater
1348 than the maximum length for the property being overridden.

1349 **5.5.2.26 MaxValue**

1350 The MaxValue qualifier takes uint32 values and has Scope(Property Parameter Method). The default
1351 value is NULL.

1352 The MaxValue qualifier specifies the maximum value of this element. MaxValue may be used only on
1353 numeric data types. If MaxValue is applied to CIM elements with a numeric array data type, it applies to
1354 every element of the array. A value of NULL means that the maximum value is the highest value for the
1355 data type.

1356 An overriding property that specifies the MaxValue qualifier must specify a maximum value no greater
1357 than the maximum value of the property being overridden.

1358 **5.5.2.27 MethodConstraint**

1359 The MethodConstraint qualifier takes string array values and has Scope(Method). The default value is
1360 NULL.

1361 The qualified element specifies one or more constraints, which are defined using the Object Constraint
1362 Language (OCL), as specified in the OMG [Object Constraint Language Specification](#).

1363 The MethodConstraint array contains string values that specify OCL precondition, postcondition, and
1364 body constraints.

1365 The OCL context of these constraints (that is, what "self" in OCL refers to) is the object on which the
1366 qualified method is invoked.

1367 An OCL precondition constraint is expressed as a typed OCL expression that specifies whether the
1368 precondition is satisfied. The type of the expression shall be Boolean. For the method to complete
1369 successfully, all preconditions of a method shall be satisfied before it is invoked.

1370 A string value specifying an OCL precondition constraint shall conform to the syntax:

1371 `ocl_precondition_string = "pre" [ocl_name] ":" ocl_statement`

1372 Where:

1373 `ocl_name` is the name of the OCL constraint.

1374 `ocl_statement` is the OCL statement of the precondition constraint, which defines the Boolean
1375 expression.

1376 An OCL postcondition constraint is expressed as a typed OCL expression that specifies whether the
1377 postcondition is satisfied. The type of the expression shall be Boolean. All postconditions of the method
1378 shall be satisfied immediately after successful completion of the method.

1379 A string value specifying an OCL post-condition constraint shall conform to the following syntax:

1380 `ocl_postcondition_string = "post" [ocl_name] ":" ocl_statement`

1381 Where:

1382 `ocl_name` is the name of the OCL constraint.

1383 `ocl_statement` is the OCL statement of the post-condition constraint, which defines the Boolean expression.

1385 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.

1388 A string value specifying an OCL body constraint shall conform to the following syntax:

1389 `ocl_body_string = "body" [ocl_name] ":" ocl_statement`

1390 Where:

1391 `ocl_name` is the name of the OCL constraint.

1392 `ocl_statement` is the OCL statement of the body constraint, which defines the method return value.

1393 EXAMPLE: The following qualifier defined on the RequestedStateChange() method of the
1394 EnabledLogicalElement class specifies that if a Job parameter is returned as not NULL, then an OwningJobElement
1395 association must exist between the EnabledLogicalElement class and the Job.

```
1396 MethodConstraint {
1397     "post AssociatedJob:"
1398     "not Job.oclIsUndefined()"
1399     "implies"
1400     "self.cIM_OwningJobElement.OwnedElement = Job"
1401 }
```

1402 5.5.2.28 Min

1403 The Min qualifier takes uint32 values and has Scope(Reference). The default value is "0".

1404 The Min qualifier specifies the minimum cardinality of the reference, which is the minimum number of
1405 values a given reference may have for each set of other reference values in the association. For example,
1406 if an association relates A instances to B instances and there shall be at least one A instance for each B
1407 instance, then the reference to A should have a Min(1) qualifier.

1408 The qualifier value shall not be NULL.

1409 5.5.2.29 MinLen

1410 The MinLen qualifier takes uint32 values and has Scope(Property Parameter Method). The default value
1411 is "0".

1412 The MinLen qualifier specifies the minimum length, in characters, of a string data item. MinLen may be
1413 used only on string data types. If MinLen is applied to CIM elements with a string array data type, it
1414 applies to every element of the array. The NULL value is not allowed for MinLen.

1415 An overriding property that specifies the MINLEN qualifier must specify a minimum length no smaller than
1416 the minimum length of the property being overridden.

1417 5.5.2.30 MinValue

1418 The MinValue qualifier takes sint64 values and has Scope(Property Parameter Method). The default
1419 value is NULL.

1420 The MinValue qualifier specifies the minimum value of this element. MinValue may be used only on
 1421 numeric data types. If MinValue is applied to CIM elements with a numeric array data type, it applies to
 1422 every element of the array. A value of NULL means that the minimum value is the lowest value for the
 1423 data type.

1424 An overriding property that specifies the MinValue qualifier must specify a minimum value no smaller than
 1425 the minimum value of the property being overridden.

1426 5.5.2.31 ModelCorrespondence

1427 The ModelCorrespondence qualifier takes string array values and has Scope(Class Association Indication
 1428 Property Reference Parameter Method). The default value is NULL.

1429 The ModelCorrespondence qualifier indicates a correspondence between two elements in the CIM
 1430 schema. The referenced elements shall be defined in a standard or extension MOF file, such that the
 1431 correspondence can be examined. If possible, forward referencing of elements should be avoided.

1432 Object elements are identified using the following syntax:

```
1433 <className> [ *("(<propertyName> | <referenceName> ) ) [ "." <methodName> [ "("
```

```
1434 <parameterName> "]" ] ] ]
```

1435 Note that the basic relationship between the referenced elements is a "loose" correspondence, which
 1436 simply indicates that the elements are coupled. This coupling may be unidirectional. Additional qualifiers
 1437 may be used to describe a tighter coupling.

1438 The following list provides examples of several correspondences found in CIM and vendor schemas:

- 1439 • A vendor defines an Indication class corresponding to a particular CIM property or method so
 1440 that Indications are generated based on the values or operation of the property or method. In
 1441 this case, the ModelCorrespondence may only be on the vendor's Indication class, which is an
 1442 extension to CIM.
- 1443 • A property provides more information for another. For example, an enumeration has an allowed
 1444 value of "Other", and another property further clarifies the intended meaning of "Other." In
 1445 another case, a property specifies status and another property provides human-readable strings
 1446 (using an array construct) expanding on this status. In these cases, ModelCorrespondence is
 1447 found on both properties, each referencing the other. Also, referenced array properties may not
 1448 be ordered but carry the default ArrayType qualifier definition of "Bag."
- 1449 • A property is defined in a subclass to supplement the meaning of an inherited property. In this
 1450 case, the ModelCorrespondence is found only on the construct in the subclass.
- 1451 • Multiple properties taken together are needed for complete semantics. For example, one
 1452 property may define units, another property may define a multiplier, and another property may
 1453 define a specific value. In this case, ModelCorrespondence is found on all related properties,
 1454 each referencing all the others.
- 1455 • Multi-dimensional arrays are desired. For example, one array may define names while another
 1456 defines the name formats. In this case, the arrays are each defined with the
 1457 ModelCorrespondence qualifier, referencing the other array properties or parameters. Also, they
 1458 are indexed and they carry the ArrayType qualifier with the value "Indexed."

1459 The semantics of the correspondence are based on the elements themselves. ModelCorrespondence is
 1460 only a hint or indicator of a relationship between the elements.

1461 5.5.2.32 NonLocal

1462 This instance-level qualifier and the corresponding pragma were removed as an erratum by CR1461.

1463 5.5.2.33 NonLocalType

1464 This instance-level qualifier and the corresponding pragma were removed as an erratum by CR1461.

1465 5.5.2.34 NullValue

1466 The NullValue qualifier takes string values and has Scope(Property). The default value is NULL.

1467 The NullValue qualifier defines a value that indicates that the associated property is NULL. That is, the
1468 property is considered to have a valid or meaningful value.

1469 The NullValue qualifier may be used only with properties that have string and integer values. When used
1470 with an integer type, the qualifier value is a MOF integer value. The syntax for representing an integer
1471 value is:

1472 ["+" / "-"] 1* <decimalDigit>

1473 The content, maximum number of digits, and represented value are constrained by the data type of the
1474 qualified property.

1475 Note that this qualifier cannot be overridden because it seems unreasonable to permit a subclass to
1476 return a different null value than that of the superclass.

1477 5.5.2.35 OctetString

1478 The OctetString qualifier takes Boolean values and has Scope(Property Parameter Method). The default
1479 value is FALSE.

1480 This qualifier identifies the qualified property or parameter as an octet string.

1481 When used in conjunction with an unsigned 8-bit integer (uint8) array, the OctetString qualifier indicates
1482 that the unsigned 8-bit integer array represents a single octet string.

1483 When used in conjunction with arrays of strings, the OctetString qualifier indicates that the qualified
1484 character strings are encoded textual conventions representing octet strings. The text encoding of these
1485 binary values conforms to the following grammar: "0x" 4*(<hexDigit> <hexDigit>). In both cases, the first 4
1486 octets of the octet string (8 hexadecimal digits in the text encoding) are the number of octets in the
1487 represented octet string with the length portion included in the octet count. (For example, "0x00000004" is
1488 the encoding of a 0 length octet string. A second example is "0x000000050A" that is an encoding of the
1489 octet string "0x0A".)

1490 5.5.2.36 Out

1491 The Out qualifier takes Boolean values and has Scope(Parameter). The default value is FALSE.

1492 The Out qualifier indicates that the associated parameter is used to return values from a method.

1493 5.5.2.37 Override

1494 The Override qualifier takes string values and has Scope(Property Parameter Method). The default value
1495 is NULL.

1496 If non-NULL, the qualified element in the derived (containing) class takes the place of another element (of
1497 the same name) defined in the ancestry of that class.

1498 The flavor of the qualifier is defined as 'Restricted' so that the Override qualifier is not repeated in
1499 (inherited by) each subclass. The effect of the override is inherited, but not the identification of the
1500 Override qualifier itself. This enables new Override qualifiers in subclasses to be easily located and
1501 applied.

1502 An effective value of NULL (the default) indicates that the element is not overriding any element. If not
 1503 NULL, the value shall have the following format:

1504 [className"."] IDENTIFIER,

1505 where IDENTIFIER shall be the name of the overridden element and if present, className shall be
 1506 the name of a class in the ancestry of the derived class. The className shall be present if the class
 1507 exposes more than one element with the same name. (See 7.5.1.)

1508 If the className is omitted, the overridden element is found by searching the ancestry of the class until a
 1509 definition of an appropriately-named subordinate element (of the same meta-schema class) is found.

1510 If the className is specified, the element being overridden is found by searching the named class and its
 1511 ancestry until a definition of an element of the same name (of the same meta-schema class) is found.

1512 The Override qualifier may only refer to elements of the same meta-schema class. For example,
 1513 properties can only override properties, etc. An element's name or signature shall not be changed when
 1514 overriding.

1515 **5.5.2.38 Propagated**

1516 The Propagated qualifier takes string values and has Scope(Property). The default value is NULL.

1517 The Propagated qualifier is a string-valued qualifier that contains the name of the key that is propagated.
 1518 Its use assumes only one Weak qualifier on a reference with the containing class as its target. The
 1519 associated property shall have the same value as the property named by the qualifier in the class on the
 1520 other side of the weak association. The format of the string to accomplish this is as follows:

1521 [<className> "."] <IDENTIFIER>

1522 When the Propagated qualifier is used, the Key qualifier shall be specified with a value of TRUE.

1523 **5.5.2.39 PropertyConstraint**

1524 The PropertyConstraint qualifier takes string array values and has Scope(Property Reference). The
 1525 default value is NULL.

1526 The qualified element specifies one or more constraints that are defined using the Object Constraint
 1527 Language (OCL) as specified in the OMG [Object Constraint Language Specification](#).

1528 The PropertyConstraint array contains string values that specify OCL initialization and derivation
 1529 constraints. The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of
 1530 the class, association, or indication that exposes the qualified property or reference.

1531 An OCL initialization constraint is expressed as a typed OCL expression that specifies the permissible
 1532 initial value for a property. The type of the expression shall conform to the CIM data type of the property.

1533 A string value specifying an OCL initialization constraint shall conform to the following syntax:

1534 ocl_initialization_string = "init" ":" ocl_statement

1535 Where:

1536 ocl_statement is the OCL statement of the initialization constraint, which defines the typed
 1537 expression.

1538 An OCL derivation constraint is expressed as a typed OCL expression that specifies the permissible
 1539 value for a property at any time in the lifetime of the instance. The type of the expression shall conform to
 1540 the CIM data type of the property.

1541 A string value specifying an OCL derivation constraint shall conform to the following syntax:

1542 `ocl_derivation_string = "derive" ":" ocl_statement`

1543 Where:

1544 `ocl_statement` is the OCL statement of the derivation constraint, which defines the typed expression.

1545 For example, PolicyAction has a SystemName property that must be set to the name of the system
1546 associated with PolicySetInSystem. The following qualifier defined on PolicyAction.SystemName specifies
1547 that constraint:

```
1548     PropertyConstraint {  
1549         "derive: self.CIM_PolicySetInSystem.Antecedent.Name"  
1550     }
```

1551 A property shall not be qualified with more than one initialization constraint or derivation constraint. The
1552 definition of an initialization constraint and a derivation constraint on the same property is allowed. In this
1553 case, the value of the property immediately after creation of the instance shall satisfy both constraints.

1554 5.5.2.40 PUnit

1555 The PUnit qualifier takes string array values and has Scope(Property Parameter Method). The default
1556 value is NULL.

1557 The PUnit qualifier indicates the programmatic unit of measure of the qualified property, method return
1558 value, or method parameter. The qualifier value follows the syntax for programmatic units.

1559 NULL indicates that the programmatic unit is unknown. The syntax for programmatic units is defined in
1560 ANNEX C.

1561 Experimental: This qualifier has a status of "Experimental."

1562 5.5.2.41 Read

1563 The Read qualifier takes Boolean values and has Scope(Property). The default value is TRUE.

1564 The Read qualifier indicates that the property is readable.

1565 5.5.2.42 Required

1566 The Required qualifier takes Boolean values and has Scope(Property Reference Parameter Method). The
1567 default value is FALSE.

1568 A non-NULL value is required for the element. For CIM elements with an array type, the Required
1569 qualifier affects the array itself, and the elements of the array may be NULL regardless of the Required
1570 qualifier.

1571 Properties of a class that are inherent characteristics of a class and identify that class are such properties
1572 as domain name, file name, burned-in device identifier, IP address, and so on. These properties are likely
1573 to be useful for applications as query entry points that are not KEY properties but should be Required
1574 properties.

1575 References of an association that are not KEY references shall be Required references. There are no
1576 particular usage rules for using the Required qualifier on parameters of a method outside of the meaning
1577 defined in this clause.

1578 A property that overrides a required property shall not specify REQUIRED(false).

1579 5.5.2.43 Revision (Deprecated)

1580 The Revision qualifier is deprecated. (See 5.5.2.53 for the description of the Version qualifier.)

1581 The Revision qualifier takes string values and has Scope(Class Association Indication). The default value
1582 is NULL.

1583 The Revision qualifier provides the minor revision number of the schema object.

1584 The Version qualifier shall be present to supply the major version number when the Revision qualifier is
1585 used.

1586 5.5.2.44 Schema (Deprecated)

1587 The Schema string qualifier is deprecated. The schema for any feature can be determined by examining
1588 the complete class name of the class defining that feature.

1589 The Schema string qualifier takes string values and has Scope(Property Method). The default value is
1590 NULL.

1591 The Schema qualifier indicates the name of the schema that contains the feature.

1592 5.5.2.45 Source

1593 This instance-level qualifier and the corresponding pragma are removed as an erratum by CR1461.

1594 5.5.2.46 SourceType

1595 This instance-level qualifier and the corresponding pragma are removed as an erratum by CR1461.

1596 5.5.2.47 Static

1597 The Static qualifier takes Boolean values and has Scope(Property Method). The default value is FALSE.

1598 The property or method is static. For a definition of static properties, see 7.5.6. For a definition of static
1599 methods, see 7.9.1.

1600 An element that overrides a non-static element shall not be a static element.

1601 5.5.2.48 Terminal

1602 The Terminal qualifier takes Boolean values and has Scope(Class Association Indication). The default
1603 value is FALSE.

1604 The class can have no subclasses. If such a subclass is declared, the compiler generates an error.

1605 This qualifier cannot coexist with the Abstract qualifier. If both are specified, the compiler generates an
1606 error.

1607 5.5.2.49 UMLPackagePath

1608 The UMLPackagePath qualifier takes string values and has Scope(Class Association Indication). The
1609 default value is NULL.

1610 This qualifier specifies a position within a UML package hierarchy for a CIM class.

1611 The qualifier value shall consist of a series of package names, each interpreted as a package within the
1612 preceding package, separated by '::'. The first package name in the qualifier value shall be the schema
1613 name of the qualified CIM class.

1614 For example, consider a class named "CIM_Abc" that is in a package named "PackageB" that is in a
1615 package named "PackageA" that, in turn, is in a package named "CIM." The resulting qualifier
1616 specification for this class "CIM_Abc" is as follows:

1617 UMLPACKAGEPATH ("CIM::PackageA::PackageB")

1618 A value of NULL indicates that the following default rule shall be used to create the UML package path:
1619 The name of the UML package path is the schema name of the class, followed by "::default".

1620 For example, a class named "CIM_Xyz" with a UMLPackagePath qualifier value of NULL has the UML
1621 package path "CIM::default".

1622 5.5.2.50 Units (Deprecated)

1623 The Units qualifier is deprecated. Instead, the PUnit qualifier should be used for programmatic access,
1624 and the client application should use its own conventions to construct a string to be displayed from the
1625 PUnit qualifier.

1626 The Units qualifier takes string values and has Scope(Property Parameter Method). The default value is
1627 NULL.

1628 The Units qualifier specifies the unit of measure of the qualified property, method return value, or method
1629 parameter. For example, a Size property might have a unit of "Bytes."

1630 NULL indicates that the unit is unknown. An empty string indicates that the qualified property, method
1631 return value, or method parameter has no unit and therefore is dimensionless. The complete set of DMTF
1632 defined values for the Units qualifier is presented in ANNEX C.

1633 5.5.2.51 ValueMap

1634 The ValueMap qualifier takes string array values and has Scope(Property Parameter Method). The
1635 default value is NULL.

1636 The ValueMap qualifier defines the set of permissible values for the qualified property, method return, or
1637 method parameter.

1638 The ValueMap qualifier can be used alone or in combination with the Values qualifier. When it is used
1639 with the Values qualifier, the location of the value in the ValueMap array determines the location of the
1640 corresponding entry in the Values array.

1641 Where:

1642 ValueMap may be used only with string or integer types.

1643 When used with a string type, a ValueMap entry is a MOF stringvalue.

1644 When used with an integer type, a ValueMap entry is a MOF integervalue or an integervalue range as
1645 defined here.

1646 integervalue range:
1647 [integervalue] ".." [integervalue]

1648 A ValueMap entry of :

1649 "x" claims the value x.

1650 "..x" claims all values less than and including x.

1651 "x.." claims all values greater than and including x.

1652 ".." claims all values not otherwise claimed.

1653 The values claimed are constrained by the type of the associated property.

1654 ValueMap = ("..") is not permitted.

1655 If used with a Value array, then all values claimed by a particular ValueMap entry apply to the
1656 corresponding Value entry.

1657 EXAMPLE:

1658 [Values {"zero&one", "2to40", "fifty", "the unclaimed", "128-255"}, ValueMap {"..1", "2..40" "50", "..", "x80.." }]
1659 uint8 example;

1660 In this example, where the type is uint8, the following mappings are made:

1661 "..1" and "zero&one" map to 0 and 1.

1662 "2..40" and "2to40" map to 2 through 40.

1663 ".." and "the unclaimed" map to 41 through 49 and to 51 through 127.

1664 "0x80.." and "128-255" map to 128 through 255.

1665 An overriding property that specifies the ValueMap qualifier shall not map any values not allowed by the
1666 overridden property. In particular, if the overridden property specifies or inherits a ValueMap qualifier,
1667 then the overriding ValueMap qualifier must map only values that are allowed by the overridden
1668 ValueMap qualifier. (Note, however, that the overriding property may organize these values differently
1669 than does the overridden property. For example, ValueMap {"0..10"} may be overridden by ValueMap
1670 {"0..1", "2..9"}.) An overriding ValueMap qualifier may specify fewer values than the overridden property
1671 would otherwise allow.

1672 5.5.2.52 Values

1673 The Values qualifier takes string array values and has Scope(Property Parameter Method). The default
1674 value is NULL.

1675 The Values qualifier translates between integer values and strings (such as abbreviations or English
1676 terms) in the ValueMap array, and an associated string at the same index in the Values array. If a
1677 ValueMap qualifier is not present, the Values array is indexed (zero relative) using the value in the
1678 associated property, method return type, or method parameter. If a ValueMap qualifier is present, the
1679 Values index is defined by the location of the property value in the ValueMap. If both Values and
1680 ValueMap are specified or inherited, the number of entries in the Values and ValueMap arrays shall
1681 match.

1682 5.5.2.53 Version

1683 The Version qualifier takes string values and has Scope(Class Association Indication). The default
1684 value is NULL.

1685 The Version qualifier provides the version information of the object, which increments when changes are
1686 made to the object.

1687 Starting with CIM Schema 2.7 (including extension schema), the Version qualifier shall be present on
1688 each class to indicate the version of the last update to the class.

1689 The string representing the version comprises three decimal integers separated by periods; that is,
1690 M.N.U, or, more formally, 1*<decimalDigit> "." 1*<decimalDigit> "." 1*<decimalDigit>

1691 The meaning of M.N.U is as follows:

1692 **M** - The major version in numeric form of the change to the class.

1693 **N** - The minor version in numeric form of the change to the class.

1694 **U** - The update (for example, errata, patch, ...) in numeric form of the change to the class.

1695 NOTE 1: The addition or removal of the Experimental qualifier does not require the version information to be
1696 updated.

1697 NOTE 2: The version change applies only to elements that are local to the class. In other words, the version change
1698 of a superclass does not require the version in the subclass to be updated.

1699 EXAMPLE:

1700 Version("2.7.0")

1701 Version("1.0.0")

1702 **5.5.2.54 Weak**

1703 The Weak qualifier takes Boolean values and has Scope(Reference). The default value is FALSE.

1704 The keys of the referenced class include the keys of the other participants in the association. This
1705 qualifier is used when the identity of the referenced class depends on that of the other participants in the
1706 association. No more than one reference to any given class can be weak. The other classes in the
1707 association shall define a key. The keys of the other classes are repeated in the referenced class and
1708 tagged with a propagated qualifier.

1709 **5.5.2.55 Write**

1710 The Write qualifier takes Boolean values and has Scope(Property). The default value is FALSE.

1711 The modeling semantics of a property support modification of that property by consumers. The purpose of
1712 this qualifier is to capture modeling semantics and not to address more dynamic characteristics such as
1713 provider capability or authorization rights.

1714 **5.5.3 Optional Qualifiers**

1715 The following subclauses list the optional qualifiers that address situations that are not common to all
1716 CIM-compliant implementations. Thus, CIM-compliant implementations can ignore optional qualifiers
1717 because they are not required to interpret or understand them. The optional qualifiers are provided in the
1718 specification to avoid random user-defined qualifiers for these recurring situations.

1719 **5.5.3.1 Alias**

1720 The Alias qualifier takes string values and has Scope(Property Reference Method). The default value is
1721 NULL.

1722 The Alias qualifier establishes an alternate name for a property or method in the schema.

1723 **5.5.3.2 Delete**

1724 The Delete qualifier takes Boolean values and has Scope(Association Reference). The default value is
1725 FALSE.

1726 **For associations:** The qualified association shall be deleted if any of the objects referenced in the
1727 association are deleted and the respective object referenced in the association is qualified with IfDeleted.

1728 **For references:** The referenced object shall be deleted if the association containing the reference is
1729 deleted and qualified with IfDeleted. It shall also be deleted if any objects referenced in the association
1730 are deleted and the respective object referenced in the association is qualified with IfDeleted.

1731 Applications shall chase associations according to the modeled semantic and delete objects
1732 appropriately.

1733 NOTE: This usage rule must be verified when the CIM security model is defined.

1734 **5.5.3.3 DisplayDescription**

1735 The DisplayDescription qualifier takes string values and has Scope(Class Association Indication Property
1736 Reference Parameter Method). The default value is NULL.

1737 The DisplayDescription qualifier defines descriptive text for the qualified element for display on a human
1738 interface — for example, fly-over Help or field Help.

1739 The DisplayDescription qualifier is for use within extension subclasses of the CIM schema to provide
1740 display descriptions that conform to the information development standards of the implementing product.
1741 A value of NULL indicates that no display description is provided. Therefore, a display description
1742 provided by the corresponding schema element of a superclass can be removed without substitution.

1743 **5.5.3.4 Expensive**

1744 The Expensive qualifier takes string values and has Scope(Class Association Indication Property
1745 Reference Parameter Method).The default value is FALSE.

1746 The Expensive qualifier indicates that the element is expensive to manipulate and/or compute.

1747 **5.5.3.5 IfDeleted**

1748 The IfDeleted qualifier takes Boolean values and has Scope(Association Reference). The default value is
1749 FALSE.

1750 All objects qualified by Delete within the association shall be deleted if the referenced object or the
1751 association, respectively, is deleted.

1752 **5.5.3.6 Invisible**

1753 The Invisible qualifier takes Boolean values and has Scope(Class Association Property Reference
1754 Method). The default value is FALSE.

1755 The Invisible qualifier indicates that the element is defined only for internal purposes and should not be
1756 displayed or otherwise relied upon. For example, an intermediate value in a calculation or a value to
1757 facilitate association semantics is defined only for internal purposes.

1758 **5.5.3.7 Large**

1759 The Large qualifier takes Boolean values and has Scope(Class Property). The default value is FALSE.

1760 The Large qualifier property or class requires a large amount of storage space.

1761 **5.5.3.8 PropertyUsage**

1762 The PropertyUsage qualifier takes string values and has Scope(Property). The default value is
1763 "CURRENTCONTEXT".

1764 This qualifier allows properties to be classified according to how they are used by managed elements.
1765 Therefore, the managed element can convey intent for property usage. The qualifier does not convey
1766 what access CIM has to the properties. That is, not all configuration properties are writeable. Some
1767 configuration properties may be maintained by the provider or resource that the managed element
1768 represents, and not by CIM. The PropertyUsage qualifier enables the programmer to distinguish between
1769 properties that represent attributes of the following:

- 1770 • A managed resource versus capabilities of a managed resource
- 1771 • Configuration data for a managed resource versus metrics about or from a managed resource
- 1772 • State information for a managed resource.

1773 If the qualifier value is set to CurrentContext (the default value), then the value of PropertyUsage should
 1774 be determined by looking at the class in which the property is placed. The rules for which default
 1775 PropertyUsage values belong to which classes/subclasses are as follows:

1776 Class>CurrentContext PropertyUsage Value
 1777 Setting > Configuration
 1778 Configuration > Configuration
 1779 Statistic > Metric ManagedSystemElement > State Product > Descriptive
 1780 FRU > Descriptive
 1781 SupportAccess > Descriptive
 1782 Collection > Descriptive

1783 The valid values for this qualifier are as follows:

- 1784 • **UNKNOWN.** The property's usage qualifier has not been determined and set.
- 1785 • **OTHER.** The property's usage is not Descriptive, Capabilities, Configuration, Metric, or State.
- 1786 • **CURRENTCONTEXT.** The PropertyUsage value shall be inferred based on the class placement
 1787 of the property according to the following rules:
 - 1788 – If the property is in a subclass of Setting or Configuration, then the PropertyUsage value of
 1789 CURRENTCONTEXT should be treated as CONFIGURATION.
 - 1790 – If the property is in a subclass of Statistics, then the PropertyUsage value of
 1791 CURRENTCONTEXT should be treated as METRIC.
 - 1792 – If the property is in a subclass of ManagedSystemElement, then the PropertyUsage value
 1793 of CURRENTCONTEXT should be treated as STATE.
 - 1794 – If the property is in a subclass of Product, FRU, SupportAccess or Collection, then the
 1795 PropertyUsage value of CURRENTCONTEXT should be treated as DESCRIPTIVE.
- 1796 • **DESCRIPTIVE.** The property contains information that describes the managed element, such
 1797 as vendor, description, caption, and so on. These properties are generally not good candidates
 1798 for representation in Settings subclasses.
- 1799 • **CAPABILITY.** The property contains information that reflects the inherent capabilities of the
 1800 managed element regardless of its configuration. These are usually specifications of a product.
 1801 For example, VideoController.MaxMemorySupported=128 is a capability.
- 1802 • **CONFIGURATION.** The property contains information that influences or reflects the
 1803 configuration state of the managed element. These properties are candidates for representation
 1804 in Settings subclasses. For example, VideoController.CurrentRefreshRate is a configuration
 1805 value.
- 1806 • **STATE** indicates that the property contains information that reflects or can be used to derive the
 1807 current status of the managed element.
- 1808 • **METRIC** indicates that the property contains a numerical value representing a statistic or metric
 1809 that reports performance-oriented and/or accounting-oriented information for the managed
 1810 element. This would be appropriate for properties containing counters such as
 1811 “BytesProcessed”.

1812 5.5.3.9 Provider

1813 The Provider qualifier takes string values and has Scope(Class Association Indication Property Reference
 1814 Parameter Method). The default value is NULL.

1815 An implementation-specific handle to the instrumentation that populates elements in the schemas that
1816 refers to dynamic data.

1817 **5.5.3.10 Syntax**

1818 The Syntax qualifier takes string values and has Scope(Property, Reference, Parameter Method). The
1819 default value is NULL.

1820 The Syntax qualifier indicates the specific type assigned to a data item. It must be used with the
1821 SyntaxType qualifier.

1822 **5.5.3.11 SyntaxType**

1823 The SyntaxType qualifier takes string values and has Scope(Property Reference Parameter Method). The
1824 default value is NULL.

1825 The SyntaxType qualifier defines the format of the Syntax qualifier. It must be used with the Syntax
1826 qualifier.

1827 **5.5.3.12 TriggerType**

1828 The TriggerType qualifier takes string values and has Scope(Class Association Indication Property
1829 Reference Method). The default value is NULL.

1830 The TriggerType qualifier specifies the circumstances that cause a trigger to be fired.

1831 The trigger types vary by meta-model construct. For classes and associations, the legal values are
1832 CREATE, DELETE, UPDATE, and ACCESS. For properties and references, the legal values are
1833 UPDATE and ACCESS. For methods, the legal values are BEFORE and AFTER. For indications, the
1834 legal value is THROWN.

1835 **5.5.3.13 UnknownValues**

1836 The UnknownValues qualifier takes string values and has Scope(Property). The default value is NULL.

1837 The UnknownValues qualifier specifies a set of values that indicates that the value of the associated
1838 property is unknown. Therefore, the property cannot be considered to have a valid or meaningful value.

1839 The conventions and restrictions for defining unknown values are the same as those for the ValueMap
1840 qualifier.

1841 The UnknownValues qualifier cannot be overridden because it is unreasonable for a subclass to treat as
1842 known a value that a superclass treats as unknown.

1843 **5.5.3.14 UnsupportedValues**

1844 The UnsupportedValues qualifier takes string values and has Scope(Property). The default value is
1845 NULL.

1846 The UnsupportedValues qualifier specifies a set of values that indicates that the value of the associated
1847 property is unsupported. Therefore, the property cannot be considered to have a valid or meaningful
1848 value.

1849 The conventions and restrictions for defining unsupported values are the same as those for the ValueMap
1850 qualifier.

1851 The UnsupportedValues qualifier cannot be overridden because it is unreasonable for a subclass to treat
1852 as supported a value that a superclass treats as unknown.

1853 5.5.4 User-defined Qualifiers

1854 The user can define any additional arbitrary named qualifiers. However, it is recommended that only
1855 defined qualifiers be used and that the list of qualifiers be extended only if there is no other way to
1856 accomplish the objective.

1857 5.5.5 Mapping Entities of Other Information Models to CIM

1858 The MappingStrings qualifier can be used to map entities of other information models to CIM or to
1859 express that a CIM element represents an entity of another information model. Several mapping string
1860 formats are defined in this clause to use as values for this qualifier. The CIM schema shall use only the
1861 mapping string formats defined in this specification. Extension schemas should use only the mapping
1862 string formats defined in this specification.

1863 The mapping string formats defined in this specification conform to the following formal syntax:

```
1864 mappingstrings_format = mib_format | oid_format | general_format | mif_format
```

1865 NOTE: As defined in the respective clauses, the "MIB", "OID", and "MIF" formats support a limited form of
1866 extensibility by allowing an open set of defining bodies. However, the syntax defined for these formats does not allow
1867 variations by defining body; they need to conform. A larger degree of extensibility is supported in the general format,
1868 where the defining bodies may define a part of the syntax used in the mapping.

1869 5.5.5.1 SNMP-Related Mapping String Formats

1870 The two SNMP-related mapping string formats, Management Information Base (MIB) and globally unique
1871 object identifier (OID), can express that a CIM element represents a MIB variable. As defined in [RFC1155](#)
1872 a MIB variable has an associated variable name that is unique within a MIB and an OID that is unique
1873 within a management protocol.

1874 The "MIB" mapping string format identifies a MIB variable using naming authority, MIB name, and variable
1875 name. It may be used only on CIM properties, parameters, or methods. The format is defined as follows:

```
1876 mib_format = "MIB" "." mib_naming_authority "|" mib_name "." mib_variable_name
```

1877 Where:

```
1878 mib_naming_authority = 1*(stringChar)
```

1879 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and
1880 vertical bar (|) characters are not allowed.

```
1881 mib_name = 1*(stringChar)
```

1882 is the name of the MIB as defined by the MIB naming authority (for example, "HOST-
1883 RESOURCES-MIB"). The dot (.) and vertical bar (|) characters are not allowed.

```
1884 mib_variable_name = 1*(stringChar)
```

1885 is the name of the MIB variable as defined in the MIB (for example, "hrSystemDate"). The dot
1886 (.) and vertical bar (|) characters are not allowed.

1887 The tokens in mib_format should be assembled without intervening white space characters. The MIB
1888 name should be the ASN.1 module name of the MIB (that is, not the RFC number). For example, instead
1889 of using "RFC1493", the string "BRIDGE-MIB" should be used.

1890 For example:

```
1891 [MappingStrings { "MIB.IETF|HOST-RESOURCES-MIB.hrSystemDate" }]
```

```
1892 datetime LocalDateTime;
```

1893 The "OID" mapping string format identifies a MIB variable using a management protocol and an object
 1894 identifier (OID) within the context of that protocol. This format is especially important for mapping
 1895 variables defined in private MIBs. It may be used only on CIM properties, parameters, or methods. The
 1896 format is defined as follows:

1897 `oid_format = "OID" "." oid_naming_authority "|" oid_protocol_name "." oid`

1898 Where:

1899 `oid_naming_authority = 1*(stringChar)`

1900 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and
 1901 vertical bar (|) characters are not allowed.

1902 `oid_protocol_name = 1*(stringChar)`

1903 is the name of the protocol providing the context for the OID of the MIB variable (for example,
 1904 "SNMP"). The dot (.) and vertical bar (|) characters are not allowed.

1905 `oid = 1*(stringChar)`

1906 is the object identifier (OID) of the MIB variable in the context of the protocol (for example,
 1907 "1.3.6.1.2.1.25.1.2").

1908 The tokens in `oid_format` should be assembled without intervening white space characters.

1909 EXAMPLE:

1910 `[MappingStrings { "OID.IETF|SNMP.1.3.6.1.2.1.25.1.2" }]`

1911 `datetime LocalDateTime;`

1912 For both mapping string formats, the name of the naming authority defining the MIB shall be one of the
 1913 following:

- 1914 • The name of a standards body (for example, IETF), for standard MIBs defined by that standards
 1915 body
- 1916 • A company name (for example, Acme), for private MIBs defined by that company

1917 5.5.5.2 General Mapping String Format

1918 This clause defines the mapping string format, which provides a basis for future mapping string formats.
 1919 Future mapping string formats defined in this document should be based on the general mapping string
 1920 format. A mapping string format based on this format shall define the kinds of CIM elements with which it
 1921 is to be used.

1922 The format is defined as follows. Note that the division between the name of the format and the actual
 1923 mapping is slightly different than for the "MIF", "MIB", and "OID" formats:

1924 `general_format = general_format_fullname "|" general_format_mapping`

1925 `general_format_fullname = general_format_name "." general_format_defining_body`

1926 Where:

1927 `general_format_name = 1*(stringChar)`

1928 is the name of the format, unique within the defining body. The dot (.) and vertical bar (|)
 1929 characters are not allowed.

1930 `general_format_defining_body = 1*(stringChar)`

1931 is the name of the defining body. The dot (.) and vertical bar (|) characters are not allowed.

1932 `general_format_mapping = 1*(stringChar)`

- 1933 is the mapping of the qualified CIM element, using the named format.
- 1934 The tokens in `general_format` and `general_format_fullname` should be assembled without intervening
1935 white space characters.
- 1936 The text in Figure 6 is an example that defines a mapping string format based on the general mapping
1937 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.

1938 **Figure 6 – Example for Mapping a String Format Based on the General Mapping String Format**

1939 5.5.5.3 MIF-Related Mapping String Format

1940 Management Information Format (MIF) attributes can be mapped to CIM elements using the
1941 MappingStrings qualifier. This qualifier maps DMTF and vendor-defined MIF groups to CIM classes or
1942 properties using either domain or recast mapping.

1943 **Deprecation Note:** MIF is defined in the DMTF *Desktop Management Interface Specification*, which
1944 completed DMTF end of life in 2005 and is therefore no longer considered relevant. Any occurrence of
1945 the MIF format in values of the MappingStrings qualifier is considered deprecated. Any other usage of
1946 MIF in this specification is also considered deprecated. The MappingStrings qualifier itself is not
1947 deprecated because it is used for formats other than MIF.

1948 As stated in the DMTF *Desktop Management Interface Specification*, every MIF group defines a unique
1949 identification that uses the MIF class string, which has the following formal syntax:

```
1950 mif_class_string = mif_defining_body "|" mif_specific_name "|" mif_version
```

1951 where:

```
1952 mif_defining_body = 1*(stringChar)
```

1953 is the name of the body defining the group. The dot (.) and vertical bar (|) characters are not
1954 allowed.

1955 `mif_specific_name = 1*(stringChar)`

1956 is the unique name of the group. The dot (.) and vertical bar (|) characters are not allowed.

1957 `mif_version = 3(decimalDigit)`

1958 is a three-digit number that identifies the version of the group definition.

1959 By default, the formal syntax rules in this (current) specification allow each token to be separated by an
 1960 arbitrary number of white spaces. However, the DMTF *Desktop Management Interface Specification*
 1961 considers MIF class strings to be opaque identification strings for MIF groups. MIF class strings that differ
 1962 only in white space characters are considered to be different identification strings.

1963 In addition, each MIF attribute has a unique numeric identifier, starting with the number one, using the
 1964 following formal syntax:

1965 `mif_attribute_id = positiveDecimalDigit *decimalDigit`

1966 A MIF domain mapping maps an individual MIF attribute to a particular CIM property. A MIF recast
 1967 mapping maps an entire MIF group to a particular CIM class.

1968 The MIF format for use as a value of the MappingStrings qualifier has the following formal syntax:

1969 `mif_format = mif_attribute_format | mif_group_format`

1970 Where:

1971 `mif_attribute_format = "MIF" "." mif_class_string "." mif_attribute_id`

1972 is used for mapping a MIF attribute to a CIM property.

1973 `mif_group_format = "MIF" "." mif_class_string`

1974 is used for mapping a MIF group to a CIM class.

1975 For example, a MIF domain mapping of a MIF attribute to a CIM property is as follows:

1976 `[MappingStrings { "MIF.DMTF|ComponentID|001.4" }]`

1977 `string SerialNumber;`

1978 A MIF recast mapping maps an entire MIF group into a CIM class, as follows:

1979 `[MappingStrings { "MIF.DMTF|Software Signature|002" }]`

1980 `class SoftwareSignature`

1981 `{`

1982 `...`

1983 `};`

1984 **6 Managed Object Format**

1985 The management information is described in a language based on [ISO/IEC 14750:1999](#) called the
 1986 Managed Object Format (MOF). In this document, the term "MOF specification" refers to a collection of
 1987 management information described in a way that conforms to the MOF syntax. Elements of MOF syntax
 1988 are introduced on a case-by-case basis with examples. In addition, a complete description of the MOF
 1989 syntax is provided in ANNEX A.

1990 NOTE: All grammars defined in this specification use the notation defined in [RFC 4234](#); any exceptions are stated
 1991 with the grammar.

1992 The MOF syntax describes object definitions in textual form and therefore establishes the syntax for
 1993 writing definitions. The main components of a MOF specification are textual descriptions of classes,

1994 associations, properties, references, methods, and instance declarations and their associated qualifiers.
1995 Comments are permitted.

1996 In addition to serving the need for specifying the managed objects, a MOF specification can be processed
1997 using a compiler. To assist the process of compilation, a MOF specification consists of a series of
1998 compiler directives.

1999 A MOF file can be encoded in either Unicode or UTF-8.

2000 **6.1 MOF Usage**

2001 The managed object descriptions in a MOF specification can be validated against an active namespace
2002 (see clause 8). Such validation is typically implemented in an entity acting in the role of a server. This
2003 clause describes the behavior of an implementation when introducing a MOF specification into a
2004 namespace. Typically, such a process validates both the syntactic correctness of a MOF specification and
2005 its semantic correctness against a particular implementation. In particular, MOF declarations must be
2006 ordered correctly with respect to the target implementation state. For example, if the specification
2007 references a class without first defining it, the reference is valid only if the server already has a definition
2008 of that class. A MOF specification can be validated for the syntactic correctness alone, in a component
2009 such as a MOF compiler.

2010 **6.2 Class Declarations**

2011 A class declaration is treated as an instruction to create a new class. Whether the process of introducing
2012 a MOF specification into a namespace can add classes or modify classes is a local matter. If the
2013 specification references a class without first defining it, the server must reject it as invalid if it does not
2014 already have a definition of that class.

2015 **6.3 Instance Declarations**

2016 Any instance declaration is treated as an instruction to create a new instance where the key values of the
2017 object do not already exist or an instruction to modify an existing instance where an object with identical
2018 key values already exists.

2019 **7 MOF Components**

2020 The following subclauses describe the components of MOF syntax.

2021 **7.1 Keywords**

2022 All keywords in the MOF syntax are case-insensitive.

2023 **7.2 Comments**

2024 Comments can appear anywhere in MOF syntax and are indicated by either a leading double slash (//)
2025 or a pair of matching /* and */ sequences.

2026 A // comment is terminated by carriage return, line feed, or the end of the MOF specification (whichever
2027 comes first).

2028 EXAMPLE:

```
2029     // This is a comment
```

2030 A /* comment is terminated by the next */ sequence or by the end of the MOF specification (whichever
2031 comes first). The meta model does not recognize comments, so they are not preserved across
2032 compilations. Therefore, the output of a MOF compilation is not required to include any comments.

2033 7.3 Validation Context

2034 Semantic validation of a MOF specification involves an explicit or implied namespace context. This is
2035 defined as the namespace against which the objects in the MOF specification are validated and the
2036 namespace in which they are created. Multiple namespaces typically indicate the presence of multiple
2037 management spaces or multiple devices.

2038 7.4 Naming of Schema Elements

2039 This clause describes the rules for naming schema elements, including classes, properties, qualifiers,
2040 methods, and namespaces.

2041 CIM is a conceptual model that is not bound to a particular implementation. Therefore, it can be used to
2042 exchange management information in a variety of ways, examples of which are described in the
2043 [Introduction](#). Some implementations may use case-sensitive technologies, while others may use case-
2044 insensitive technologies. The naming rules defined in this clause allow efficient implementation in either
2045 environment and enable the effective exchange of management information among all compliant
2046 implementations.

2047 All names are case-insensitive, so two schema item names are identical if they differ only in case. This is
2048 mandated so that scripting technologies that are case-insensitive can leverage CIM technology. However,
2049 string values assigned to properties and qualifiers are not covered by this rule and must be treated as
2050 case-sensitive.

2051 The case of a name is set by its defining occurrence and must be preserved by all implementations. This
2052 is mandated so that implementations can be built using case-sensitive technologies such as Java and
2053 object databases. This also allows names to be consistently displayed using the same user-friendly
2054 mixed-case format. For example, an implementation, if asked to create a Disk class must reject the
2055 request if there is already a DISK class in the current schema. Otherwise, when returning the name of the
2056 Disk class it must return the name in mixed case as it was originally specified.

2057 CIM does not currently require support for any particular query language. It is assumed that
2058 implementations will specify which query languages are supported by the implementation and will adhere
2059 to the case conventions that prevail in the specified language. That is, if the query language is case-
2060 insensitive, statements in the language will behave in a case-insensitive way.

2061 For the full rules for schema names, see ANNEX E.

2062 7.5 Class Declarations

2063 A class is an object describing a grouping of data items that are conceptually related and that model an
2064 object. Class definitions provide a type system for instance construction.

2065 7.5.1 Declaring a Class

2066 A class is declared by specifying these components:

- 2067 • Qualifiers of the class, which can be empty, or a list of qualifier name/value bindings separated
2068 by commas (,) and enclosed with square brackets ([and]).
- 2069 • Class name.
- 2070 • Name of the class from which this class is derived, if any.
- 2071 • Class properties, which define the data members of the class. A property may also have an
2072 optional qualifier list expressed in the same way as the class qualifier list. In addition, a property
2073 has a data type, and (optionally) a default (initializer) value.

- 2074 • Methods supported by the class. A method may have an optional qualifier list, and it has a
2075 signature consisting of its return type plus its parameters and their type and usage.
- 2076 • A CIM class may expose more than one element (property or method) with a given name, but it
2077 is not permitted to define more than one element with a particular name. This can happen if a
2078 base class defines an element with the same name as an element defined in a derived class
2079 without overriding the base class element. (Although considered rare, this could happen in a
2080 class defined in a vendor extension schema that defines a property or method that uses the
2081 same name that is later chosen by an addition to an ancestor class defined in the common
2082 schema.)

2083 This sample shows how to declare a class:

```

2084     [abstract]
2085     class Win32_LogicalDisk
2086     {
2087         [read]
2088         string DriveLetter;
2089         [read, Units("KiloBytes")]
2090         sint32 RawCapacity = 0;
2091         [write]
2092         string VolumeLabel;
2093         [Dangerous]
2094         boolean Format([in] boolean FastFormat);
2095     };

```

2096 7.5.2 Subclasses

2097 To indicate that a class is a subclass of another class, the derived class is declared by using a colon
2098 followed by the superclass name. For example, if the class Acme_Disk_v1 is derived from the class
2099 CIM_Media:

```

2100     class Acme_Disk_v1 : CIM_Media
2101     {
2102         // Body of class definition here ...
2103     };

```

2104 The terms base class, superclass, and supertype are used interchangeably, as are derived class,
2105 subclass, and subtype. The superclass declaration must appear at a prior point in the MOF specification
2106 or already be a registered class definition in the namespace in which the derived class is defined.

2107 7.5.3 Default Property Values

2108 Any properties in a class definition can have default initializers. For example:

```

2109     class Acme_Disk_v1 : CIM_Media
2110     {
2111         string Manufacturer = "Acme";
2112         string ModelNumber = "123-AAL";
2113     };

```

2114 When new instances of the class are declared, any such property is automatically assigned its default
2115 value unless the instance declaration explicitly assigns a value to the property.

2116 7.5.4 Class and Property Qualifiers

2117 Qualifiers are meta data about a property, method, method parameter, or class, and they are not part of
2118 the definition itself. For example, a qualifier indicates whether a property value can be changed (using the
2119 Write qualifier). Qualifiers always precede the declaration to which they apply.

2120 Certain qualifiers are well known and cannot be redefined (see 5.5). Apart from these restrictions,
2121 arbitrary qualifiers may be used.

2122 Qualifier declarations include an explicit type indicator, which must be one of the intrinsic types. A
2123 qualifier with an array-based parameter is assumed to have a type, which is a variable-length
2124 homogeneous array of one of the intrinsic types. In Boolean arrays, each element in the array is either
2125 TRUE or FALSE.

2126 EXAMPLE:

```
2127     Write(true)                // boolean
2128     profile { true, false, true } // boolean []
2129     description("A string")     // string
2130     info { "this", "a", "bag", "is" } // string []
2131     id(12)                      // uint32
2132     idlist { 21, 22, 40, 43 }   // uint32 []
2133     apple(3.14)                // real32
2134     oranges { -1.23E+02, 2.1 } // real32 []
```

2135 Qualifiers are applied to a class by preceding the class declaration with a qualifier list, comma-separated
2136 and enclosed within square brackets. Qualifiers are applied to a property or method in a similar way.

2137 EXAMPLE:

```
2138     class CIM_Process: CIM_LogicalElement
2139     {
2140         uint32 Priority;
2141         [Write(true)]
2142         string Handle;
2143     };
```

2144 When a Boolean qualifier is specified in a class or property declaration, the name of the qualifier can be
2145 used without also specifying a value. From the previous example:

```
2146     class CIM_Process: CIM_LogicalElement
2147     {
2148         uint32 Priority;
2149         [Write] // Equivalent declaration to Write (True)
2150         string Handle;
2151     };
```

2152 If only the qualifier name is listed for a Boolean qualifier, it is implicitly set to TRUE. In contrast, when a
2153 qualifier is not specified at all for a class or property, the default value for the qualifier is assumed.
2154 Consider another example:

```
2155         [Association,
2156         Aggregation] // Specifies the Aggregation qualifier to be True
2157     class CIM_SystemDevice: CIM_SystemComponent
2158     {
2159         [Override ("GroupComponent"),
2160         Aggregate] // Specifies the Aggregate qualifier to be True
2161         CIM_ComputerSystem Ref GroupComponent;
2162         [Override ("PartComponent"),
2163         Weak] // Defines the Weak qualifier to be True
2164         CIM_LogicalDevice Ref PartComponent;
2165     };
2166
2167     [Association] // Since the Aggregation qualifier is not specified,
2168                 // its default value, False, is set
2169     class Acme_Dependency: CIM_Dependency
2170     {
2171         [Override ("Antecedent")] // Since the Aggregate and Weak
2172                                   // qualifiers are not used, their
2173                                   // default values, False, are assumed
```

```
2174         Acme_SpecialSoftware Ref Antecedent;  
2175         [Override ("Dependent")]  
2176         Acme_Device Ref Dependent;  
2177     };
```

2178 Qualifiers can automatically be transmitted from classes to derived classes or from classes to instances,
2179 subject to certain rules. The rules prescribing how the transmission occurs are attached to each qualifier
2180 and encapsulated in the concept of the qualifier flavor. For example, a qualifier can be designated in the
2181 base class as automatically transmitted to all of its derived classes, or it can be designated as belonging
2182 specifically to that class and not transmittable. The former is achieved by using the ToSubclass flavor,
2183 and the latter by using the Restricted flavor. These two flavors shall not be used at the same time. In
2184 addition, if a qualifier is transmitted to its derived classes, the qualifier flavor can be used to control
2185 whether derived classes can override the qualifier value or whether the qualifier value must be fixed for
2186 an entire class hierarchy. This aspect of qualifier flavor is referred to as override permissions.

2187 Override permissions are assigned using the EnableOverride or DisableOverride flavors, which shall not
2188 be used at the same time. If a qualifier is not transmitted to its derived classes, these two flavors are
2189 meaningless and shall be ignored.

2190 Qualifier flavors are indicated by an optional clause after the qualifier and are preceded by a colon. They
2191 consist of some combination of the key words EnableOverride, DisableOverride, ToSubclass, and
2192 Restricted, indicating the applicable propagation and override rules.

2193 EXAMPLE:

```
2194     class CIM_Process: CIM_LogicalElement  
2195     {  
2196         uint32 Priority;  
2197         [Write(true):DisableOverride ToSubclass]  
2198         string Handle;  
2199     };
```

2200 In this example, Handle is designated as writable for the Process class and for every subclass of this
2201 class.

2202 The recognized flavor types are shown in Table 5.

2203

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	<p>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:</p> <p style="padding-left: 40px;">label_ll_cc</p> <p>where</p> <ul style="list-style-type: none"> ▪ 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. <p>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.</p> <p>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.</p> <p>EXAMPLE: If an English description is translated into Mexican Spanish, the actual name of the qualifier is: DESCRIPTION_es_MX.</p>	No

2204 **7.5.5 Key Properties**

2205 Instances of a class require a way to distinguish the instances within a single namespace. Designating
 2206 one or more properties with the reserved Key qualifier provides instance identification. For example, this
 2207 class has one property (Volume) that serves as its key:

```

2208     class Acme_Drive
2209     {
2210         [key]
2211         string Volume;
2212         string FileSystem;
2213         sint32 Capacity;
2214     };
    
```

2215 In this example, instances of Drive are distinguished using the Volume property, which acts as the key for
 2216 the class.

2217 Compound keys are supported and are designated by marking each of the required properties with the
 2218 key qualifier.

2219 If a new subclass is defined from a superclass and the superclass has key properties (including those
 2220 inherited from other classes), the new subclass *cannot* define any additional key properties. New key

2221 properties in the subclass can be introduced only if all classes in the inheritance chain of the new
2222 subclass are keyless.

2223 If any reference to the class has the Weak qualifier, the properties that are qualified as Key in the other
2224 classes in the association are propagated to the referenced class. The key properties are duplicated in
2225 the referenced class using the name of the property, prefixed by the name of the original declaring class.
2226 For example:

```

2227     class CIM_System: CIM_LogicalElement
2228     {
2229         [Key]
2230         string Name;
2231     };

2232     class CIM_LogicalDevice: CIM_LogicalElement
2233     {
2234         [Key]
2235         string DeviceID;
2236         [Key, Propagated("CIM_System.Name")]
2237         string SystemName;
2238     };

2239     [Association]
2240     class CIM_SystemDevice: CIM_SystemComponent
2241     {
2242         [Override("GroupComponent"), Aggregate, Min(1), Max(1)]
2243         CIM_System Ref GroupComponent;
2244         [Override("PartComponent"), Weak]
2245         CIM_LogicalDevice Ref PartComponent;
2246     };

```

2247 7.5.6 Static Properties

2248 If a property is declared as a static property, it has the same value for all CIM instances that have the
2249 property in the same namespace. Therefore, any change in the value of a static property for a CIM
2250 instance also affects the value of that property for the other CIM instances that have it. As for any
2251 property, a change in the value of a static property of a CIM instance in one namespace may or may not
2252 affect its value in CIM instances in other namespaces.

2253 Overrides on static properties are prohibited. Overrides of static methods are allowed.

2254 7.6 Association Declarations

2255 An association is a special kind of class describing a link between other classes. Associations also
2256 provide a type system for instance constructions. Associations are just like other classes with a few
2257 additional semantics, which are explained in the following subclauses.

2258 7.6.1 Declaring an Association

2259 An association is declared by specifying these components:

- 2260 • Qualifiers of the association (at least the Association qualifier, if it does not have a supertype).
2261 Further qualifiers may be specified as a list of qualifier/name bindings separated by commas
2262 (,). The entire qualifier list is enclosed in square brackets ([and]).
- 2263 • Association name. The name of the association from which this association derives (if any).
- 2264 • Association references. Define pointers to other objects linked by this association. References
2265 may also have qualifier lists that are expressed in the same way as the association qualifier list

2266 — especially the qualifiers to specify cardinalities of references (see 5.5.2). In addition, a
2267 reference has a data type, and (optionally) a default (initializer) value.

2268 • Additional association properties that define further data members of this association. They are
2269 defined in the same way as for ordinary classes.

2270 • The methods supported by the association. They are defined in the same way as for ordinary
2271 classes.

2272 EXAMPLE: The following example shows how to declare an association (assuming given classes CIM_A and
2273 CIM_B):

```
2274     [Association]
2275     class CIM_LinkBetweenAandB : CIM_Dependency
2276     {
2277         [Override ("Antecedent")]
2278         CIM_A Ref Antecedent;
2279         [Override ("Dependent")]
2280         CIM_B Ref Dependent;
2281     };
```

2282 7.6.2 Subassociations

2283 To indicate a subassociation of another association, the same notation as for ordinary classes is used.
2284 The derived association is declared using a colon followed by the superassociation name. (An example is
2285 provided in 7.6.2.)

2286 7.6.3 Key References and Properties

2287 Instances of an association also must provide a way to distinguish the instances, for they are just a
2288 special kind of a class. Designating one or more references/properties with the reserved Key qualifier
2289 identifies the instances.

2290 A reference/property of an association is (part of) the association key if the Key qualifier is applied.

```
2291     [Association, Aggregation]
2292     class CIM_Component
2293     {
2294         [Aggregate, Key]
2295         CIM_ManagedSystemElement Ref GroupComponent;
2296         [Key]
2297         CIM_ManagedSystemElement Ref PartComponent;
2298     };
```

2299 The key definition of association follows the same rules as for ordinary classes. Compound keys are
2300 supported in the same way. Also a new subassociation *cannot* define additional key
2301 properties/references. If any reference to a class has the Weak qualifier, the KEY-qualified properties of
2302 the other class, whose reference is not Weak-qualified, are propagated to the class (see 7.5.5).

2303 7.6.4 Object References

2304 Object references are special properties whose values are links or pointers to other objects (classes or
2305 instances). The value of an object reference is expressed as a string, which represents a path to another
2306 object. A non-NULL value of an object reference includes:

- 2307 • The namespace in which the object resides
- 2308 • The class name of the object
- 2309 • The values of all key properties for an instance if the object represents an instance

2310 The data type of an object reference is declared as "XXX ref", indicating a strongly typed reference to
 2311 objects of the class with name "XXX" or a derivation of this class. For example:

```
2312     [Association]
2313     class Acme_ExampleAssoc
2314     {
2315         Acme_AnotherClass ref Inst1;
2316         Acme_Aclass       ref Inst2;
2317     };
```

2318 In this declaration, Inst1 can be set to point only to instances of type Acme_AnotherClass, including
 2319 instances of its subclasses.

2320 References in associations shall not have the special NULL value.

2321 Also, see 7.12.2 for information about initializing references using aliases.

2322 In associations, object references have cardinalities that are denoted using the Min and Max qualifiers.
 2323 Examples of UML cardinality notations and their respective combinations of Min and Max values are
 2324 shown in Table 6.

2325

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

2326 7.7 Qualifier Declarations

2327 Qualifiers may be declared using the keyword "qualifier." The declaration of a qualifier allows the
 2328 definition of types, default values, propagation rules (also known as Flavors), and restrictions on use.

2329 The default value for a declared qualifier is used when the qualifier is not explicitly specified for a given
 2330 schema element. Explicit specification includes inherited qualifier specification.

2331 The MOF syntax allows a qualifier to be specified without an explicit value. The assumed value depends
 2332 on the qualifier type: Boolean types are TRUE, numeric types are NULL, strings are NULL, and arrays are
 2333 empty. For example, the Alias qualifier is declared as follows:

```
2334     qualifier alias :string = null, scope(property, reference, method);
```

2335 This declaration establishes a qualifier called alias of type string. It has a default value of NULL and may
 2336 be used only with properties, references, and methods.

2337 The meta qualifiers are declared as follows:

```
2338     Qualifier Association : boolean = false,
2339         Scope(class, association), Flavor(DisableOverride);
2340
2341     Qualifier Indication : boolean = false,
2342         Scope(class, indication), Flavor(DisableOverride);
```


2343 7.8 Instance Declarations

2344 Instances are declared using the keyword sequence "instance of" and the class name. The property
2345 values of the instance may be initialized within an initialization block. Any qualifiers specified for the
2346 instance shall already be present in the defining class and shall have the same value and flavors.

2347 Property initialization consists of an optional list of preceding qualifiers, the name of the property, and an
2348 optional value. Any qualifiers specified for the property shall already be present in the property definition
2349 from the defining class, and they shall have the same value and flavors. Any property values not
2350 initialized have default values as specified in the class definition, or (if no default value is specified) the
2351 special value NULL to indicate absence of value. For example, given the class definition:

```
2352     class Acme_LogicalDisk: CIM_Partition
2353     {
2354         [key]
2355         string DriveLetter;
2356         [Units("kilo bytes")]
2357         sint32 RawCapacity = 128000;
2358         [write]
2359         string VolumeLabel;
2360         [Units("kilo bytes")]
2361         sint32 FreeSpace;
2362     };
```

2363 an instance of this class can be declared as follows:

```
2364     instance of Acme_LogicalDisk
2365     {
2366         DriveLetter = "C";
2367         VolumeLabel = "myvol";
2368     };
```

2369 The resulting instance takes these property values:

- 2370 • DriveLetter is assigned the value "C".
- 2371 • RawCapacity is assigned the default value 128000.
- 2372 • VolumeLabel is assigned the value "myvol".
- 2373 • FreeSpace is assigned the value NULL.

2374 For subclasses, all properties in the superclass must have their values initialized along with the properties
2375 in the subclass. Any property values not specifically assigned in the instance block have either the default
2376 value for the property (if there is one) or the value NULL.

2377 The values of all key properties must be specified for an instance to be identified and created. There is no
2378 requirement to initialize other property values explicitly. See 7.11.6 for information on behavior when
2379 there is no property value initialization.

2380 As described in item 21)-e) of 5.1, a class may have, by inheritance, more than one property with a
2381 particular name. If a property initialization has a property name that is scoped to more than one property
2382 in the class, the initialization applies to the property defined closest to the class of the instance. That is,
2383 the property can be located by starting at the class of the instance. If the class defines a property with the
2384 name from the initialization, then that property is initialized. Otherwise, the search is repeated from the
2385 direct superclass of the class. See ANNEX I for more information about the name conflict issue.

2386 Instances of associations may also be defined, as in the following example:

```
2387     instance of CIM_ServiceSAPDependency
2388     {
2389         Dependent = "CIM_Service.Name = \"mail\"";
2390         Antecedent = "CIM_ServiceAccessPoint.Name = \"PostOffice\"";
2391     };
```

2392 7.8.1 Instance Aliasing

2393 An alias can be assigned to an instance using this syntax:

```
2394     instance of Acme_LogicalDisk as $Disk
2395     {
2396         // Body of instance definition here ...
2397     };
```

2398 Such an alias can later be used within the same MOF specification as a value for an object reference
2399 property. For more information, see 7.12.2.

2400 7.8.2 Arrays

2401 Arrays of any of the basic data types can be declared in the MOF specification by using square brackets
2402 after the property or parameter identifier. If there is an unsigned integer constant within the square
2403 brackets, the array is a fixed-length array and the constant indicates the size of the array; if there is
2404 nothing within the square brackets, the array is a variable-length array. Otherwise, the array definition is
2405 invalid.

2406 Fixed-length arrays always have the specified number of elements. Elements cannot be added to or
2407 deleted from fixed-length arrays, but the values of elements can be changed.

2408 Variable-length arrays have a number of elements between 0 and an implementation-defined maximum.
2409 Elements can be added to or deleted from variable-length array properties, and the values of existing
2410 elements can be changed.

2411 Element addition, deletion, or modification is defined only for array properties because array parameters
2412 are only transiently instantiated when a CIM method is invoked. For array parameters, the array is
2413 thought to be created by the CIM client for input parameters and by the CIM server side for output
2414 parameters. The array is thought to be retrieved and deleted by the CIM server side for input parameters
2415 and by the CIM client for output parameters.

2416 Array indexes start at 0 and have no gaps throughout the entire array, both for fixed-length and variable-
2417 length arrays. The special NULL value signifies the absence of a value for an element, not the absence of
2418 the element itself. In other words, array elements that are NULL exist in the array and have a value of
2419 NULL. They do not represent gaps in the array.

2420 Like any CIM type, an array itself may have the special NULL value to indicate absence of value.
2421 Conceptually, the value of the array itself, if not absent, is the set of its elements. An empty array (that is,
2422 an array with no elements) must be distinguishable from an array that has the special NULL value. For
2423 example, if an array contains error messages, it makes a difference to know that there are no error
2424 messages rather than to be uncertain about whether there are any error messages.

2425 The type of an array is defined by the ArrayType qualifier with values of Bag, Ordered, or Indexed. The
2426 default array type is Bag.

2427 For a Bag array type, no significance is attached to the array index other than its convenience for
2428 accessing the elements of the array. There can be no assumption that the same index returns the same
2429 element for every retrieval, even if no element of the array is changed. The only valid assumption is that a
2430 retrieval of the entire array contains all of its elements and the index can be used to enumerate the
2431 complete set of elements within the retrieved array. The Bag array type should be used in the CIM

2432 schema when the order of elements in the array does not have a meaning. There is no concept of
2433 corresponding elements between Bag arrays.

2434 For an Ordered array type, the CIM server side maintains the order of elements in the array as long as no
2435 array elements are added, deleted, or changed. Therefore, the CIM server side does not honor any order
2436 of elements presented by the CIM client when creating the array (during creation of the CIM instance for
2437 an array property or during CIM method invocation for an input array parameter) or when modifying the
2438 array. Instead, the CIM server side itself determines the order of elements on these occasions and
2439 therefore possibly reorders the elements. The CIM server side then maintains the order it has determined
2440 during successive retrievals of the array. However, as soon as any array elements are added, deleted, or
2441 changed, the server side again determines a new order and from then on maintains that new order. For
2442 output array parameters, the server side determines the order of elements and the client side sees the
2443 elements in that same order upon retrieval. The Ordered array type should be used when the order of
2444 elements in the array does have a meaning and should be controlled by the CIM server side. The order
2445 the CIM server side applies is implementation-defined unless the class defines particular ordering rules.
2446 Corresponding elements between Ordered arrays are those that are retrieved at the same index.

2447 For an Indexed array type, the array maintains the reliability of indexes so that the same index returns the
2448 same element for successive retrievals. Therefore, particular semantics of elements at particular index
2449 positions can be defined. For example, in a status array property, the first array element might represent
2450 the major status and the following elements represent minor status modifications. Consequently, element
2451 addition and deletion is not supported for this array type. The Indexed array type should be used when
2452 the relative order of elements in the array has a meaning and should be controlled by the CIM client, and
2453 reliability of indexes is needed. Corresponding elements between Indexed arrays are those at the same
2454 index.

2455 The current release of CIM does not support n-dimensional arrays.

2456 Arrays of any basic data type are legal for properties. Arrays of references are not legal for properties.
2457 Arrays must be homogeneous; arrays of mixed types are not supported. In MOF, the data type of an
2458 array precedes the array name. Array size, if fixed-length, is declared within square brackets after the
2459 array name. For a variable-length array, empty square brackets follow the array name.

2460 Arrays are declared using the following MOF syntax:

```
2461 class A
2462 {
2463     [Description("An indexed array of variable length"), ArrayType("Indexed")]
2464     uint8 MyIndexedArray[];
2465     [Description("A bag array of fixed length")]
2466     uint8 MyBagArray[17];
2467 };
```

2468 If default values are to be provided for the array elements, this syntax is used:

```
2469 class A
2470 {
2471     [Description("A bag array property of fixed length")]
2472     uint8 MyBagArray[17] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};
2473 };
```

2474 The following MOF presents further examples of Bag, Ordered, and Indexed array declarations:

```
2475 class Acme_Example
2476 {
2477     char16 Prop1[];           // Bag (default) array of chars, Variable length
2478
2479     [ArrayType ("Ordered")] // Ordered array of double-precision reals,
2480     real64 Prop2[];         // Variable length
2481
2482     [ArrayType ("Bag")]     // Bag array containing 4 32-bit signed integers
2483     sint32 Prop3[4];
2484 }
```

```

2485     [ArrayType ("Ordered")] // Ordered array of strings, Variable length
2486     string Prop4[] = {"an", "ordered", "list"};
2487
2488     // Prop4 is variable length with default values defined at the
2489     // first three positions in the array
2490
2491     [ArrayType ("Indexed")] // Indexed array of 64-bit unsigned integers
2492     uint64 Prop5[];
2493 };

```

2494 7.9 Method Declarations

2495 A method is defined as an operation with a signature that consists of a possibly empty list of parameters
 2496 and a return type. There are no restrictions on the type of parameters other than they shall be a fixed- or
 2497 variable-length array of one of the data types described in 5.2. Method return types defined in MOF must
 2498 be one of the data types described in 5.2. Return types cannot be arrays but are otherwise unrestricted.

2499 Methods are expected, but not required, to return a status value indicating the result of executing the
 2500 method. Methods may use their parameters to pass arrays.

2501 Syntactically, the only thing that distinguishes a method from a property is the parameter list. The fact that
 2502 methods are expected to have side-effects is outside the scope of this specification.

2503 In the following example, Start and Stop methods are defined on the Service class. Each method returns
 2504 an integer value:

```

2505     class CIM_Service:CIM_LogicalElement
2506     {
2507         [Key]
2508         string Name;
2509         string StartMode;
2510         boolean Started;
2511         uint32 StartService();
2512         uint32 StopService();
2513     };

```

2514 In the following example, a Configure method is defined on the Physical DiskDrive class. It takes a
 2515 DiskPartitionConfiguration object reference as a parameter and returns a Boolean value:

```

2516     class Acme_DiskDrive:CIM_Media
2517     {
2518         sint32 BytesPerSector;
2519         sint32 Partitions;
2520         sint32 TracksPerCylinder;
2521         sint32 SectorsPerTrack;
2522         string TotalCylinders;
2523         string TotalTracks;
2524         string TotalSectors;
2525         string InterfaceType;
2526         boolean Configure([IN] DiskPartitionConfiguration REF config);
2527     };

```

2528 7.9.1 Static Methods

2529 If a method is declared as a static method, it does not depend on any per-instance data. Non-static
 2530 methods are invoked in the context of an instance; for static methods, the context of a class is sufficient.
 2531 Overrides on static properties are prohibited. Overrides of static methods are allowed.

2532 **7.10 Compiler Directives**

2533 Compiler directives are provided as the keyword "pragma" preceded by a hash (#) character and
 2534 followed by a string parameter. The current standard compiler directives are listed in Table 7.

2535 **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()	

2536 Pragma directives may be added as a MOF extension mechanism. Unless standardized in a future CIM
 2537 infrastructure specification, such new pragma definitions must be considered vendor-specific. Use of non-
 2538 standard pragma affects the interoperability of MOF import and export functions.

2539 **7.11 Value Constants**

2540 The constant types supported in the MOF syntax are described in the subclauses that follow. These are
 2541 used in initializers for classes and instances and in the parameters to named qualifiers.

2542 For a formal specification of the representation, see ANNEX A.

2543 **7.11.1 String Constants**

2544 A string constant is a sequence of zero or more UCS-2 characters enclosed in double-quotes ("). A
 2545 double-quote is allowed within the value, as long as it is preceded immediately by a backslash (\).

2546 For example, the following is a string constant:

```
2547 "This is a string"
```

2548 Successive quoted strings are concatenated as long as only white space or a comment intervenes:

```
2549 "This" " becomes a long string"
```

```
2550 "This" /* comment */ " becomes a long string"
```

2551 Escape sequences are recognized as legal characters within a string. The complete set of escape
 2552 sequences is as follows:

```

2553     \b           // \x0008: backspace BS
2554     \t           // \x0009: horizontal tab HT
2555     \n           // \x000A: linefeed LF
2556     \f           // \x000C: form feed FF
2557     \r           // \x000D: carriage return CR
2558     \"           // \x0022: double quote "
2559     \'           // \x0027: single quote '
2560     \\           // \x005C: backslash \
2561     \x<hex>     // where <hex> is one to four hex digits
2562     \X<hex>     // where <hex> is one to four hex digits

```

2563 The character set of the string depends on the character set supported by the local installation. While the
 2564 MOF specification may be submitted in UCS-2 form defined in [ISO/IEC 10646:2003](#), the local
 2565 implementation may only support ANSI and *vice versa*. Therefore, the string type is unspecified and
 2566 dependent on the character set of the MOF specification itself. If a MOF specification is submitted using
 2567 UCS-2 characters outside the normal ASCII range, the implementation may have to convert these
 2568 characters to the locally-equivalent character set.

2569 7.11.2 Character Constants

2570 Character and wide-character constants are specified as follows:

```

2571     'a'
2572     '\n'
2573     '1'
2574     '\x32'

```

2575 Forms such as octal escape sequences (for example, '\020') are not supported. Integer values can also
 2576 be used as character constants, as long as they are within the numeric range of the character type. For
 2577 example, wide-character constants must fall within the range of 0 to 0xFFFF.

2578 7.11.3 Integer Constants

2579 Integer constants may be decimal, binary, octal, or hexadecimal. For example, the following constants are
 2580 all legal:

```

2581     1000
2582     -12310
2583     0x100
2584     01236
2585     100101B

```

2586 Note that binary constants have a series of 1 and 0 digits, with a "b" or "B" suffix to indicate that the value
 2587 is binary.

2588 The number of digits permitted depends on the current type of the expression. For example, it is not legal
 2589 to assign the constant 0xFFFF to a property of type uint8.

2590 7.11.4 Floating-Point Constants

2591 Floating-point constants are declared as specified by [ANSI/IEEE 754-1985](#). For example, the following
 2592 constants are legal:

```

2593     3.14
2594     -3.14
2595     -1.2778E+02

```

2596 The range for floating-point constants depends on whether float or double properties are used, and they
2597 must fit within the range specified for 4-byte and 8-byte floating-point values, respectively.

2598 7.11.5 Object Reference Constants

2599 Object references are simple URL-style links to other objects, which may be classes or instances. They
2600 take the form of a quoted string containing an object path that is a combination of a namespace path and
2601 the model path. For example:

```
2602     "///./root/default:LogicalDisk.SystemName=\"acme\",LogicalDisk.Drive=\"C\" "
2603     "///./root/default:NetworkCard=2"
```

2604 An object reference can also be an alias. See 7.12.2 for details.

2605 7.11.6 NULL

2606 All types can be initialized to the predefined constant NULL, which indicates that no value is provided.
2607 The details of the internal implementation of the NULL value are not mandated by this document.

2608 7.12 Initializers

2609 Initializers are used in both class declarations for default values and instance declarations to initialize a
2610 property to a value. The format of initializer values is specified in clause 5 and its subclauses. The
2611 initializer value shall match the property data type. The only exceptions are the NULL value, which may
2612 be used for any data type, and integral values, which are used for characters.

2613 7.12.1 Initializing Arrays

2614 Arrays can be defined to be of type Bag, Ordered, or Indexed, and they can be initialized by specifying
2615 their values in a comma-separated list (as in the C programming language). The list of array elements is
2616 delimited with curly brackets. For example, given this class definition:

```
2617     class Acme_ExampleClass
2618     {
2619         [ArrayType ("Indexed")]
2620         string ip_addresses []; // Indexed array of variable length
2621         sint32 sint32_values [10]; // Bag array of fixed length = 10
2622     };
```

2623 the following is a valid instance declaration:

```
2624     instance of Acme_ExampleClass
2625     {
2626         ip_addresses = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };
2627
2628         // ip_address is an indexed array of at least 3 elements, where
2629         // values have been assigned to the first three elements of the
2630         // array
2631
2632         sint32_values = { 1, 2, 3, 5, 6 };
2633     };
```

2634 Refer to 7.8.2 for additional information on declaring arrays and the distinctions between bags, ordered
2635 arrays, and indexed arrays.

2636 7.12.2 Initializing References Using Aliases

2637 Aliases are symbolic references to an object located elsewhere in the MOF specification. They have
2638 significance only within the MOF specification in which they are defined, and they are used only at
2639 compile time to establish references. They are not available outside the MOF specification.

2640 An instance may be assigned an alias as described in 7.8.1. Aliases are identifiers that begin with the \$
2641 symbol. When a subsequent reference to the instance is required for an object reference property, the
2642 identifier is used in place of an explicit initializer.

2643 Assuming that \$Alias1 and \$Alias2 are declared as aliases for instances and the obref1 and obref2
2644 properties are object references, this example shows how the object references could be assigned to
2645 point to the aliased instances:

```
2646     instance of Acme_AnAssociation
2647     {
2648         strVal = "ABC";
2649         obref1 = $Alias1;
2650         obref2 = $Alias2;
2651     };
```

2652 Forward-referencing and circular aliases are permitted.

2653 8 Naming

2654 Because CIM is not bound to a particular technology or implementation, it promises to facilitate sharing
2655 management information among a variety of management platforms. The CIM naming mechanism
2656 addresses enterprise-wide identification of objects, as well as sharing of management information. CIM
2657 naming addresses the following requirements:

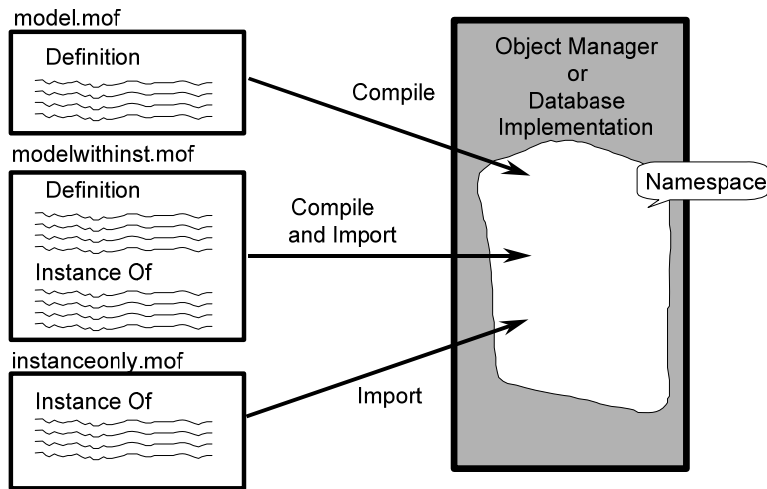
- 2658 • Ability to locate and uniquely identify any object in an enterprise. Object names must be
2659 identifiable regardless of the instrumentation technology.
- 2660 • Unambiguous enumeration of all objects.
- 2661 • Ability to determine when two object names reference the same entity. This entails location
2662 transparency so that there is no need to understand which management platforms proxy the
2663 instrumentation of other platforms.
- 2664 • Allow sharing of objects and instance data among management platforms. This requirement
2665 includes the creation of different scoping hierarchies that vary by time (for example, a current
2666 versus proposed scoping hierarchy).
- 2667 • Facilitate move operations between object trees (including within a single management
2668 platform). Hide underlying management technology/provide technology transparency for the
2669 domain-mapping environment.

2670 Allowing different names for DMI versus SNMP objects requires the management platform to understand
2671 how the underlying objects are implemented.

2672 The Key qualifier is the CIM Meta-Model mechanism to identify the properties that uniquely identify an
2673 instance of a class (and indirectly an instance of an association). CIM naming enhances this base
2674 capability by introducing the Weak and Propagated qualifiers to express situations in which the keys of
2675 one object are to be propagated to another object.

2676 **8.1 Background**

2677 CIM MOF files can contain definitions of instances, classes, or both, as illustrated in Figure 7.



2678

2679 **Figure 7 – Definitions of Instances and Classes**

2680 MOF files can be used to populate a technology that understands the semantics and structure of CIM.
 2681 When an implementation consumes a MOF, two operations are actually performed, depending on the
 2682 file's content. First, a compile or definition operation establishes the structure of the model. Second, an
 2683 import operation inserts instances into the platform or tool.

2684 When the compile and import are complete, the actual instances are manipulated using the native
 2685 capabilities of the platform or tool. To manipulate an object (for example, change the value of a property),
 2686 one must know the type of platform into which the information was imported, the APIs or operations used
 2687 to access the imported information, and the name of the platform instance actually imported. For
 2688 example, the semantics become:

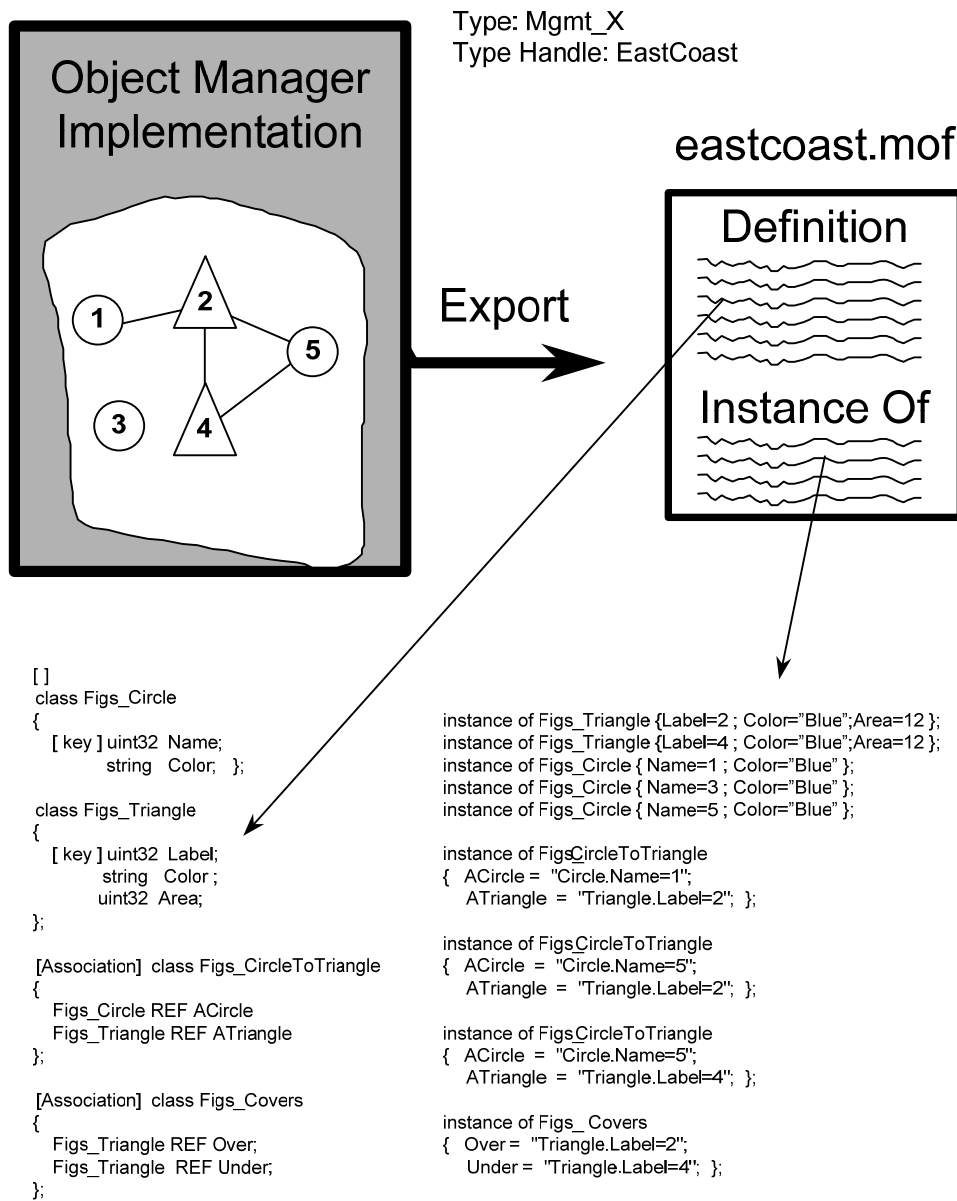
2689 Set the Version property of the Logical Element object with Name="Cool" in the relational
 2690 database named LastWeeksData to "1.4.0".

2691 The contents of a MOF file are loaded into a namespace that provides a domain in which the instances of
 2692 the classes are guaranteed to be unique per the Key qualifier definitions. The term "namespace" refers to
 2693 an implementation that provides such a domain.

2694 Namespaces can be used to accomplish the following tasks:

- 2695 • Define chunks of management information (objects and associations) to limit implementation
 2696 resource requirements, such as database size
- 2697 • Define views on the model for applications managing only specific objects, such as hubs
- 2698 • Pre-structure groups of objects for optimized query speed

2699 Another viable operation is exporting from a particular management platform. This operation creates a
 2700 MOF file for all or some portion of the information content of a platform (see Figure 8).

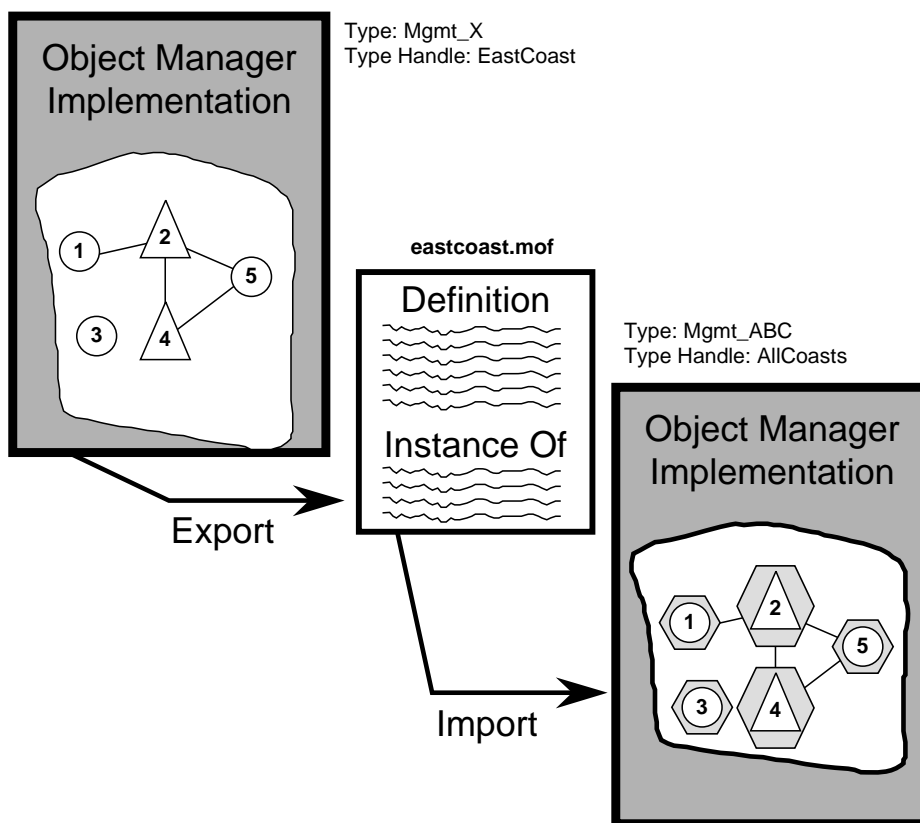


2701

2702

Figure 8 – Exporting to MOF

2703 See Figure 9 for an example. In this example, information is exchanged when the source system is of type
 2704 Mgmt_X and its name is EastCoast. The export produces a MOF file with the circle and triangle
 2705 definitions and instances 1, 3, 5 of the circle class and instances 2, 4 of the triangle class. This MOF file is
 2706 then compiled and imported into the management platform of type Mgmt_ABC with the name AllCoasts.



2707

2708

Figure 9 – Information Exchange

2709 The import operation stores the information in a local or native format of Mgmt_ABC, so its native
 2710 operations can be used to manipulate the instances. The transformation to a native format is shown in the
 2711 figure by wrapping the five instances in hexagons. The transformation process must maintain the original
 2712 keys.

2713 **8.1.1 Management Tool Responsibility for an Export Operation**

2714 The management tool must be able to create unique key values for each distinct object it places into the
 2715 MOF file. For each instance placed into the MOF file, the management tool must maintain a mapping from
 2716 the MOF file keys to the native key mechanism.

2717 **8.1.2 Management Tool Responsibility for an Import Operation**

2718 The management tool must be able to map the unique keys found in the MOF file to a set of locally-
 2719 understood keys.

2720 **8.2 Weak Associations: Supporting Key Propagation**

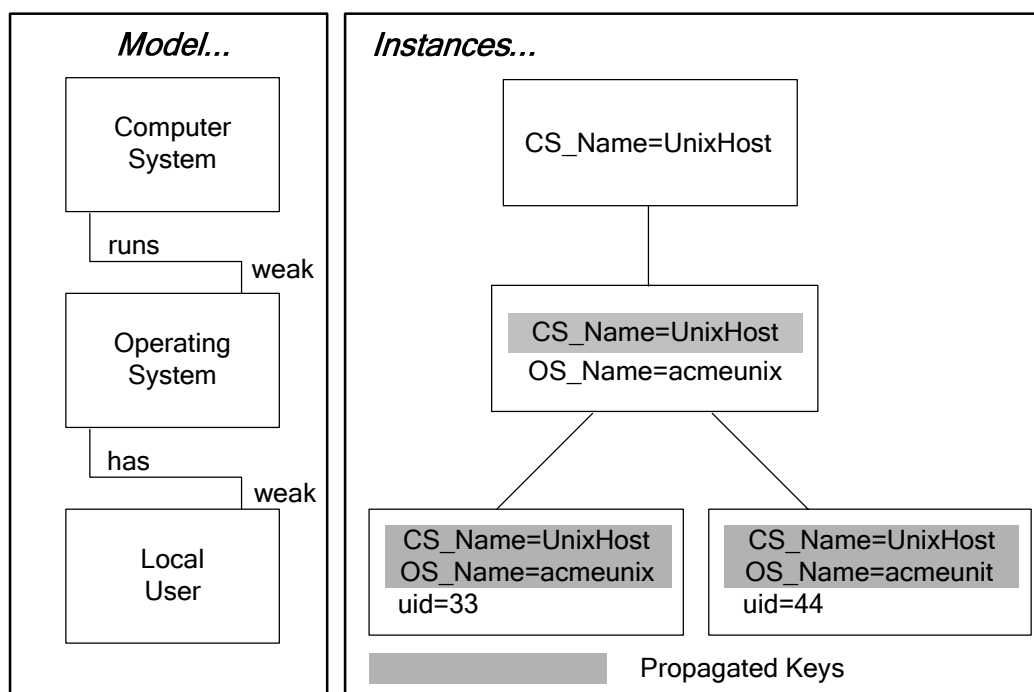
2721 CIM provides a mechanism to name instances within the context of other object instances. For example, if
 2722 a management tool handles a local system, it can refer to the C drive or the D drive. However, if a
 2723 management tool handles multiple machines, it must refer to the C drive on machine X and the C drive on
 2724 machine Y. In other words, the name of the drive must include the name of the hosting machine. CIM
 2725 supports the notion of weak associations to specify this type of key propagation. A weak association is
 2726 defined using a qualifier.

2727 EXAMPLE:

2728 `Qualifier Weak: boolean = false, Scope(reference), Flavor(DisableOverride);`

2729 The keys of the referenced class include the keys of the other participants in the Weak association. This
 2730 situation occurs when the referenced class identity depends on the identity of other participants in the
 2731 association. This qualifier can be specified on only one of the references defined for an association. The
 2732 weak referenced object is the one that depends on the other object for identity.

2733 Figure 10 shows an example of a weak association. There are three classes: ComputerSystem,
 2734 OperatingSystem and Local User. The Operating System class is weak with respect to the Computer
 2735 System class because the runs association is marked weak. Similarly, the Local User class is weak with
 2736 respect to the Operating System class, because the association is marked as weak.



2737

2738

Figure 10 – Example of Weak Association

2739 In a weak association definition, the Computer System class is a scoping class for the Operating System
 2740 class because its keys are propagated to the Operating System class. The Computer System and the
 2741 Operating System classes are both scoping classes for the Local User class because the Local User
 2742 class gets keys from both. Finally, the Computer System is referred to as a top-level object (TLO)
 2743 because it is not weak with respect to any other class. That a class is a top-level object is implied
 2744 because no references to that class are marked with the Weak qualifier. In addition, TLOs must have the
 2745 possibility of an enterprise-wide, unique key. For example, consider a computer's IP address in a

2746 company's enterprise-wide IP network. The goal of the TLO concept is to achieve uniqueness of keys in
 2747 the model path portion of the object name. To come as close as possible to this goal, the TLO must have
 2748 relevance in an enterprise context.

2749 An object in the scope of another object can in turn be a scope for a different object. Therefore, all model
 2750 object instances are arranged in directed graphs with the TLOs as peer roots. The structure of this graph,
 2751 which defines which classes are in the scope of another given class, is part of CIM by means of
 2752 associations qualified with the Weak qualifier.

2753 8.2.1 Referencing Weak Objects

2754 A reference to an instance of an association includes the propagated keys. The properties must have the
 2755 propagated qualifier that identifies the class in which the property originates and the name of the property
 2756 in that class. For example:

```
2757     instance of Acme_has
2758     {
2759         anOS = "Acme_OS.Name=\"acmeunit\",SystemName=\"UnixHost\"";
2760         aUser = "Acme_User.uid=33,OSName=\"acmeunit\",SystemName=\"UnixHost\"";
2761     };
```

2762 The operating system being weak to system is declared as follows:

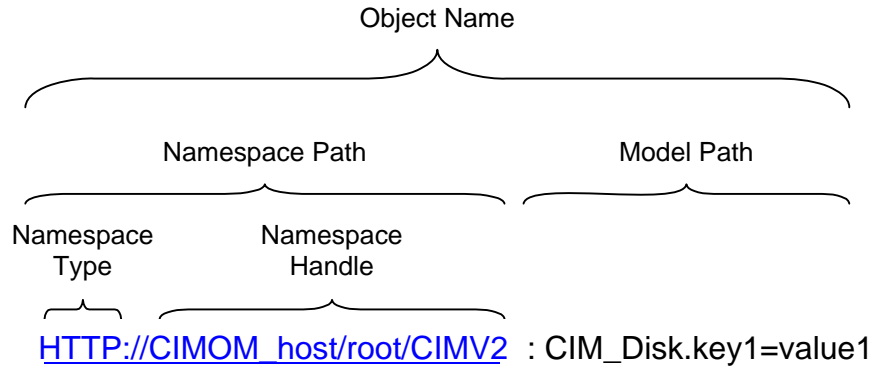
```
2763     Class Acme_OS
2764     {
2765         [key]
2766         String Name;
2767         [key, Propagated("CIM_System.Name")]
2768         String SystemName;
2769     };
```

2770 The user class being weak to operating system is declared as follows:

```
2771     Class Acme_User
2772     {
2773         [key]
2774         String uid;
2775         [key, Propagated("Acme_OS.Name")]
2776         String OSName;
2777         [key, Propagated("Acme_OS.SystemName")]
2778         String SystemName;
2779     };
```

2780 8.3 Naming CIM Objects

2781 Because CIM allows multiple implementations, it is not sufficient to think of the name of an object as just
 2782 the combination of properties that have the Key qualifier. The name must also identify the implementation
 2783 that actually hosts the objects. The object name consists of the namespace path, which provides access
 2784 to a CIM implementation, plus the model path, which provides full navigation within the CIM schema. The
 2785 namespace path is used to locate a particular namespace. The details of the namespace path depend on
 2786 the implementation. The model path is the concatenation of the class name and the properties of the
 2787 class that are qualified with the Key qualifier. When the class is weak with respect to another class, the
 2788 model path includes all key properties from the scoping objects. Figure 11 shows the various components
 2789 of an object name. These components are described in more detail in the following clauses. See the
 2790 objectName non-terminal in ANNEX A for the formal description of object name syntax.



2791

2792

Figure 11 – Object Naming**2793 8.3.1 Namespace Path**

2794 A namespace path references a namespace within an implementation that can host CIM objects. A
 2795 namespace path resolves to a namespace hosted by a CIM-capable implementation (in other words, a
 2796 CIM object manager). Unlike in the model path, the details of the namespace path are implementation-
 2797 specific. Therefore, the namespace path identifies the following details:

- 2798 • the implementation or namespace type
- 2799 • a handle that references a particular implementation or namespace handle

2800 8.3.1.1 Namespace Type

2801 The namespace type classifies or identifies the type of implementation. The provider of the
 2802 implementation must describe the access protocol for that implementation, which is analogous to
 2803 specifying http or ftp in a browser.

2804 Fundamentally, a namespace type implies an access protocol or API set to manipulate objects. These
 2805 APIs typically support the following operations:

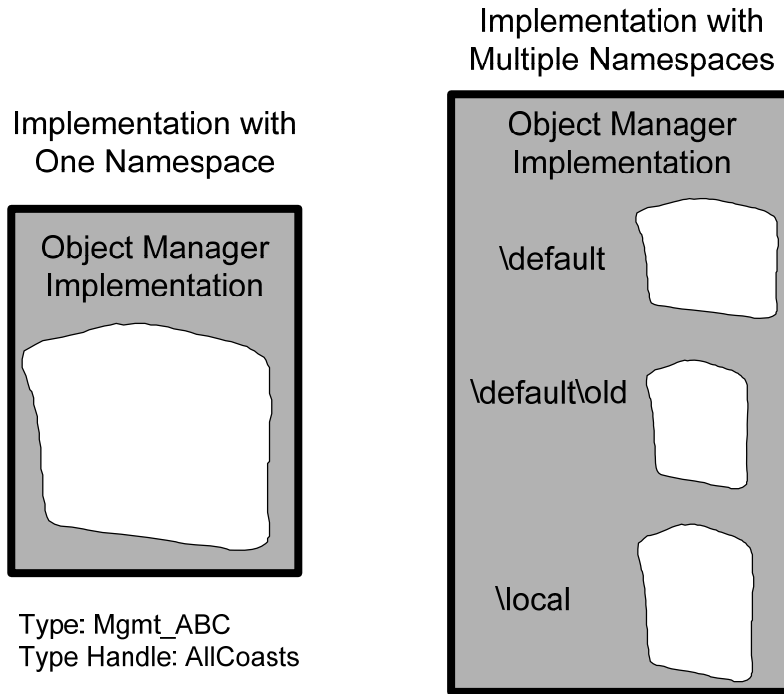
- 2806 • generating a MOF file for a particular scope of classes and associations
- 2807 • importing a MOF file
- 2808 • manipulating instances

2809 A particular management platform can access management information in a variety of ways. Each way
 2810 must have a namespace type definition. Given this type, there is an assumed set of mechanisms for
 2811 exporting, importing, and updating instances.

2812 8.3.1.2 Namespace Handle

2813 The namespace handle identifies a particular instance of the type of implementation. This handle must
 2814 resolve to a namespace within an implementation. The details of the handle are implementation-specific.
 2815 It might be a simple string for an implementation that supports one namespace, or it might be a
 2816 hierarchical structure if an implementation supports multiple namespaces. Either way, it resolves to a
 2817 namespace.

2818 Some implementations can support multiple namespaces. In this case, the implementation-specific
 2819 reference must resolve to a particular namespace within that implementation (see Figure 12).



2820

2821

Figure 12 – Namespaces

2822 Two important points to remember about namespaces are as follows:

- 2823 • Namespaces can overlap with respect to their contents.
- 2824 • When an object in one namespace has the same model path as an object in another
 2825 namespace, this does not guarantee that the objects are representing the same reality.

2826 **8.3.2 Model Path**

2827 The object name constructed as a scoping path through the CIM schema is called a model path. A model
 2828 path for an instance is a combination of the key property names and values qualified by the class name. It
 2829 is solely described by CIM elements and is absolutely implementation-independent. It can describe the
 2830 path to a particular object or to identify a particular object within a namespace. The name of any instance
 2831 is a concatenation of named key property values, including all key values of its scoping objects. When the
 2832 class is weak with respect to another class, the model path includes all key properties from the scoping
 2833 objects.

2834 The formal syntax of model path is provided in ANNEX A.

2835 The syntax of model path is as follows:

2836 `<className>.<key1>=<value1>[,<keyx>=<valuex>]*`

2837 **8.3.3 Specifying the Object Name**

2838 There are various ways to specify the object name details for any class instance or association reference
2839 in a MOF file.

2840 The model path is specified differently for objects and associations. For objects (instances of classes), the
2841 model path is the combination of property value pairs marked with the Key qualifier. Therefore, the model
2842 path for the following example is: "ex_sampleClass.label1=9921,label2=8821". Because the order of the
2843 key properties is not significant, the model path can also be: "ex_sampleClass.label2=8821,label1=9921".

```
2844     Class ex_sampleClass
2845     {
2846         [key]
2847         uint32 label1;
2848         [key]
2849         string label2;
2850         uint32 size;
2851         uint32 weight;
2852     };

2853
2854     instance of ex_sampleClass
2855     {
2856         label1 = 9921;
2857         label2 = "SampleLabel";
2858         size = 80;
2859         weight = 45
2860     };

2861
2862     instance of ex_sampleClass
2863     {
2864         label1 = 0121;
2865         label2 = "Component";
2866         size = 80;
2867         weight = 45
2868     };
```

2869 For associations, a model path specifies the value of a reference in an INSTANCE OF statement for an
2870 association. In the following composedof-association example, the model path
2871 "ex_sampleClass.label1=9921,label2=8821" references an instance of the ex_sampleClass that is playing
2872 the role of a composer:

```
2873     [Association ]
2874     Class ex_composedof
2875     {
2876         [key] composer REF ex_sampleClass;
2877         [key] component REF ex_sampleClass;
2878     };
2879     instance of ex_composedof
2880     {
2881         composer = "ex_sampleClass.label1=9921,label2=\"SampleLabel\"";
2882         component = "ex_sampleClass.label1=0121,label2=\"Component\"";
2883     }
```

2884 An object path for the ex_composedof instance is as follows. Notice how double quote characters are
2885 handled:

```
2886     ex_composedof.composer="ex_sampleClass.label1=9921,label2=\"SampleLabel\" ", componen
2887     t="ex_sampleClass.label1=0121,label2=\"Component\""
```


2888 Even in the unusual case of a reference to an association, the object name is formed the same way:

```

2889     [Association]
2890     Class ex_moreComposed
2891     {
2892         composedof REF ex_composedof;
2893         . . .
2894     };
2895
2896     instance of ex_moreComposed
2897     {
2898         composedof =
2899         "ex_composedof.composer=\"ex_sampleClass.label1=9921,label2=\\\\"SampleLabel\\\\"
2900         \",component=\"ex_sampleClass.label1=0121,label2=\\\\"Component\\\\"\"";
2901         . . .
2902     };
    
```

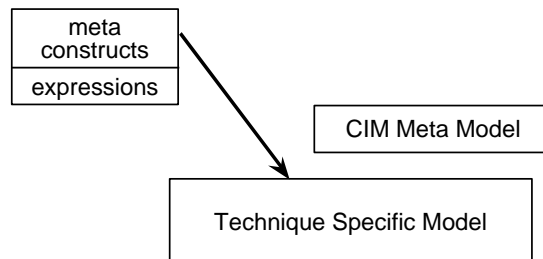
2903 The object name can be used as the value for object references and for object queries.

2904 9 Mapping Existing Models into CIM

2905 Existing models have their own meta model and model. Three types of mappings can occur between
 2906 meta schemas: technique, recast, and domain. Each mapping can be applied when MIF syntax is
 2907 converted to MOF syntax.

2908 9.1 Technique Mapping

2909 A technique mapping uses the CIM meta-model constructs to describe the meta constructs of the source
 2910 modeling technique (for example, MIF, GDMO, and SMI). Essentially, the CIM meta model is a meta
 2911 meta-model for the source technique (see Figure 13).

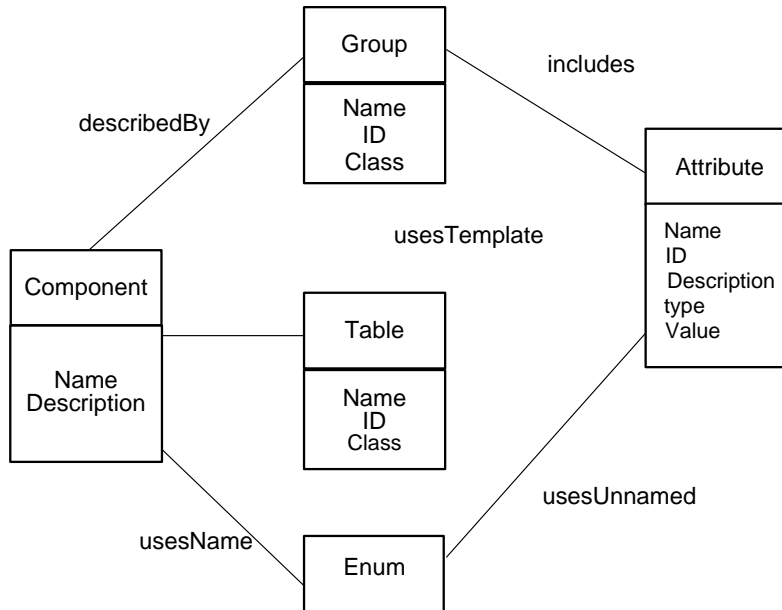


2912

2913 **Figure 13 – Technique Mapping Example**

2914 The DMTF uses the management information format (MIF) as the meta model to describe distributed
 2915 management information in a common way. Therefore, it is meaningful to describe a technique mapping
 2916 in which the CIM meta model is used to describe the MIF syntax.

2917 The mapping presented here takes the important types that can appear in a MIF file and then creates
 2918 classes for them. Thus, component, group, attribute, table, and enum are expressed in the CIM meta
 2919 model as classes. In addition, associations are defined to document how these classes are combined.
 2920 Figure 14 illustrates the results.



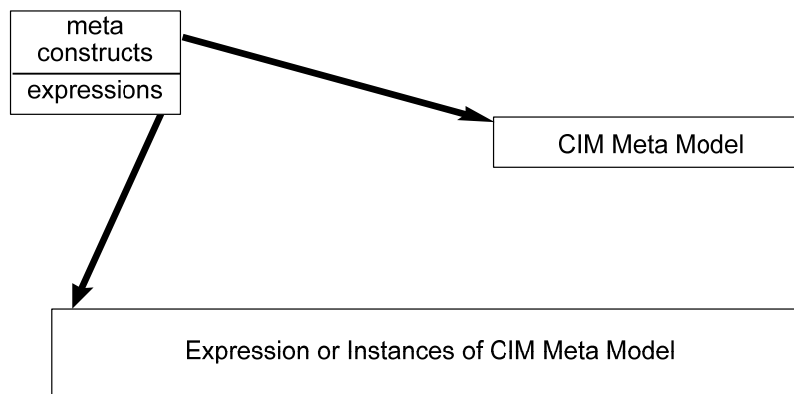
2921

2922

Figure 14 – MIF Technique Mapping Example

2923 **9.2 Recast Mapping**

2924 A recast mapping maps the meta constructs of the sources into the targeted meta constructs so that a
 2925 model expressed in the source can be translated into the target (Figure 15). The major design work is to
 2926 develop a mapping between the meta model of the sources and the CIM meta model. When this is done,
 2927 the source expressions are recast.



2928

2929

Figure 15 – Recast Mapping

2930 Following is an example of a recast mapping for MIF, assuming the following mapping:

```
2931     DMI attributes -> CIM properties
2932     DMI key attributes -> CIM key properties
2933     DMI groups -> CIM classes
2934     DMI components -> CIM classes
```

2935 The standard DMI ComponentID group can be recast into a corresponding CIM class:

```
2936     Start Group
2937     Name = "ComponentID"
2938     Class = "DMTF|ComponentID|001"
2939     ID = 1
2940     Description = "This group defines the attributes common to all "
2941                 "components. This group is required."
2942     Start Attribute
2943         Name = "Manufacturer"
2944         ID = 1
2945         Description = "Manufacturer of this system."
2946         Access = Read-Only
2947         Storage = Common
2948         Type = DisplayString(64)
2949         Value = ""
2950     End Attribute
2951     Start Attribute
2952         Name = "Product"
2953         ID = 2
2954         Description = "Product name for this system."
2955         Access = Read-Only
2956         Storage = Common
2957         Type = DisplayString(64)
2958         Value = ""
2959     End Attribute
2960     Start Attribute
2961         Name = "Version"
2962         ID = 3
2963         Description = "Version number of this system."
2964         Access = Read-Only
2965         Storage = Specific
2966         Type = DisplayString(64)
2967         Value = ""
2968     End Attribute
2969     Start Attribute
2970         Name = "Serial Number"
2971         ID = 4
2972         Description = "Serial number for this system."
2973         Access = Read-Only
2974         Storage = Specific
2975         Type = DisplayString(64)
2976         Value = ""
2977     End Attribute
2978     Start Attribute
2979         Name = "Installation"
2980         ID = 5
2981         Description = "Component installation time and date."
2982         Access = Read-Only
2983         Storage = Specific
2984         Type = Date
2985         Value = ""
2986     End Attribute
2987     Start Attribute
2988         Name = "Verify"
2989         ID = 6
2990         Description = "A code that provides a level of verification that the "
```

```

2991         "component is still installed and working."
2992     Access = Read-Only
2993     Storage = Common
2994     Type = Start ENUM
2995         0 = "An error occurred; check status code."
2996         1 = "This component does not exist."
2997         2 = "Verification is not supported."
2998         3 = "Reserved."
2999         4 = "This component exists, but the functionality is untested."
3000         5 = "This component exists, but the functionality is unknown."
3001         6 = "This component exists, and is not functioning correctly."
3002         7 = "This component exists, and is functioning correctly."
3003     End ENUM
3004     Value = 1
3005 End Attribute
3006 End Group

```

3007 A corresponding CIM class might be the following. Notice that properties in the example include an ID
3008 qualifier to represent the ID of the corresponding DMI attribute. Here, a user-defined qualifier may be
3009 necessary:

```

3010     [Name ("ComponentID"), ID (1), Description (
3011         "This group defines the attributes common to all components. "
3012         "This group is required.")]
3013     class DMTF|ComponentID|001 {
3014         [ID (1), Description ("Manufacturer of this system."), maxlen (64)]
3015         string Manufacturer;
3016         [ID (2), Description ("Product name for this system."), maxlen (64)]
3017         string Product;
3018         [ID (3), Description ("Version number of this system."), maxlen (64)]
3019         string Version;
3020         [ID (4), Description ("Serial number for this system."), maxlen (64)]
3021         string Serial_Number;
3022         [ID (5), Description("Component installation time and date.")]
3023         datetime Installation;
3024         [ID (6), Description("A code that provides a level of verification "
3025             "that the component is still installed and working."),
3026             Value (1)]
3027         string Verify;
3028     };

```

3029 9.3 Domain Mapping

3030 A domain mapping takes a source expressed in a particular technique and maps its content into either the
3031 core or common models or extension sub-schemas of the CIM. This mapping does not rely heavily on a
3032 meta-to-meta mapping; it is primarily a content-to-content mapping. In one case, the mapping is actually a
3033 re-expression of content in a more common way using a more expressive technique.

3034 Following is an example of how DMI can supply CIM properties using information from the DMI disks
3035 group ("DMTF|Disks|002"). For a hypothetical CIM disk class, the CIM properties are expressed as shown
3036 in Table 8.

3037 **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"

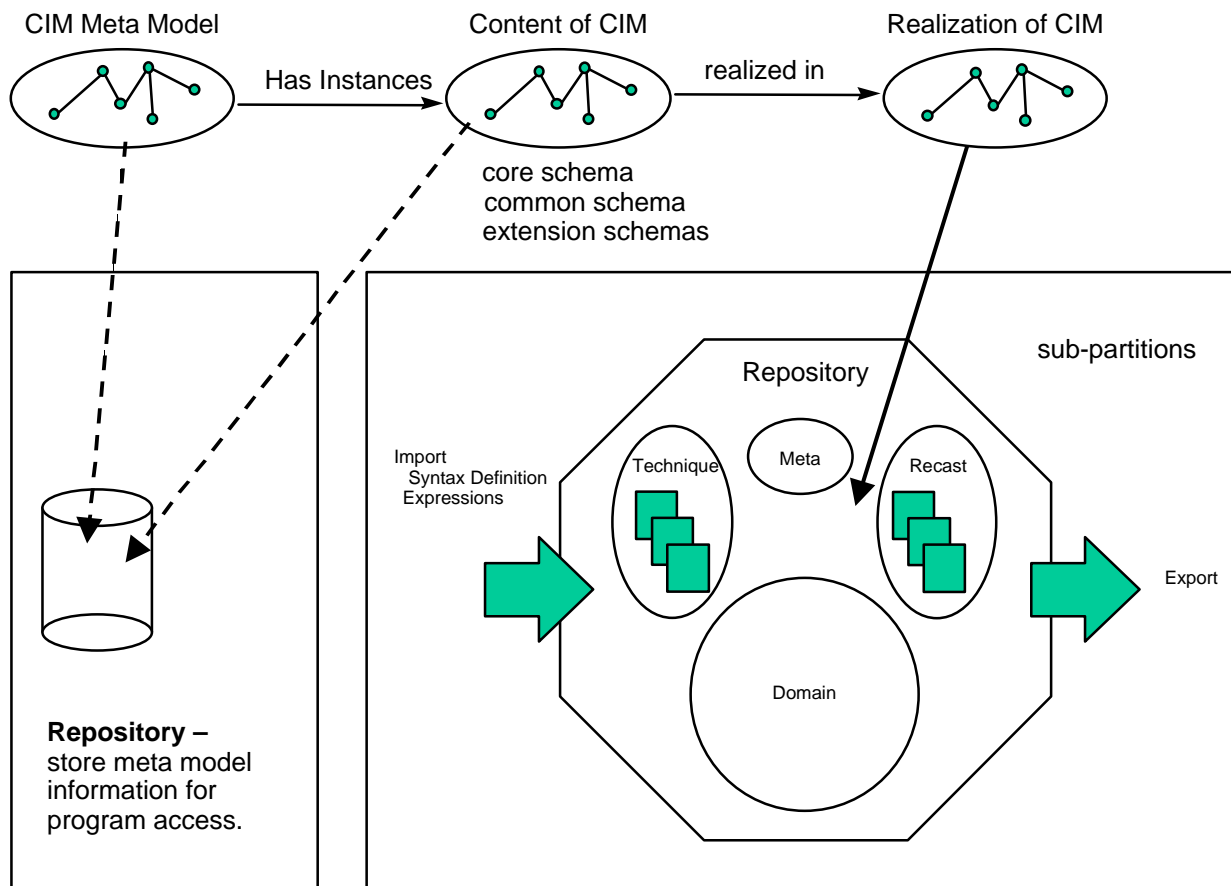
3038 **9.4 Mapping Scratch Pads**

3039 In general, when the contents of models are mapped between different meta schemas, information is lost
 3040 or missing. To fill this gap, scratch pads are expressed in the CIM meta model using qualifiers, which are
 3041 actually extensions to the meta model (for example, see 10.2). These scratch pads are critical to the
 3042 exchange of core, common, and extension model content with the various technologies used to build
 3043 management applications.

3044 **10 Repository Perspective**

3045 This clause describes a repository and presents a complete picture of the potential to exploit it. A
 3046 repository stores definitions and structural information, and it includes the capability to extract the
 3047 definitions in a form that is useful to application developers. Some repositories allow the definitions to be
 3048 imported into and exported from the repository in multiple forms. The notions of importing and exporting
 3049 can be refined so that they distinguish between three types of mappings.

3050 Using the mapping definitions in 9, the repository can be organized into the four partitions: meta,
 3051 technique, recast, and domain (see Figure 16).



3052

3053

Figure 16 – Repository Partitions

3054 The repository partitions have the following characteristics:

- 3055 • Each partition is discrete:
 - 3056 – The meta partition refers to the definitions of the CIM meta model.
 - 3057 – The technique partition refers to definitions that are loaded using technique mappings.
 - 3058 – The recast partition refers to definitions that are loaded using recast mappings.
 - 3059 – The domain partition refers to the definitions associated with the core and common models
3060 and the extension schemas.
- 3061 • The technique and recast partitions can be organized into multiple sub-partitions to capture
3062 each source uniquely. For example, there is a technique sub-partition for each unique meta
3063 language encountered (that is, one for MIF, one for GDMO, one for SMI, and so on). In the re-
3064 cast partition, there is a sub-partition for each meta language.
- 3065 • The act of importing the content of an existing source can result in entries in the recast or
3066 domain partition.

3067 10.1 DMTF MIF Mapping Strategies

3068 When the meta-model definition and the baseline for the CIM schema are complete, the next step is to
3069 map another source of management information (such as standard groups) into the repository. The main
3070 goal is to do the work required to import one or more of the standard groups. The possible import
3071 scenarios for a DMTF standard group are as follows:

- 3072 • *To Technique Partition:* Create a technique mapping for the MIF syntax that is the same for all
3073 standard groups and needs to be updated only if the MIF syntax changes.
- 3074 • *To Recast Partition:* Create a recast mapping from a particular standard group into a sub-
3075 partition of the recast partition. This mapping allows the entire contents of the selected group to
3076 be loaded into a sub-partition of the recast partition. The same algorithm can be used to map
3077 additional standard groups into that same sub-partition.
- 3078 • *To Domain Partition:* Create a domain mapping for the content of a particular standard group
3079 that overlaps with the content of the CIM schema.
- 3080 • *To Domain Partition:* Create a domain mapping for the content of a particular standard group
3081 that does not overlap with CIM schema into an extension sub-schema.
- 3082 • *To Domain Partition:* Propose extensions to the content of the CIM schema and then create a
3083 domain mapping.

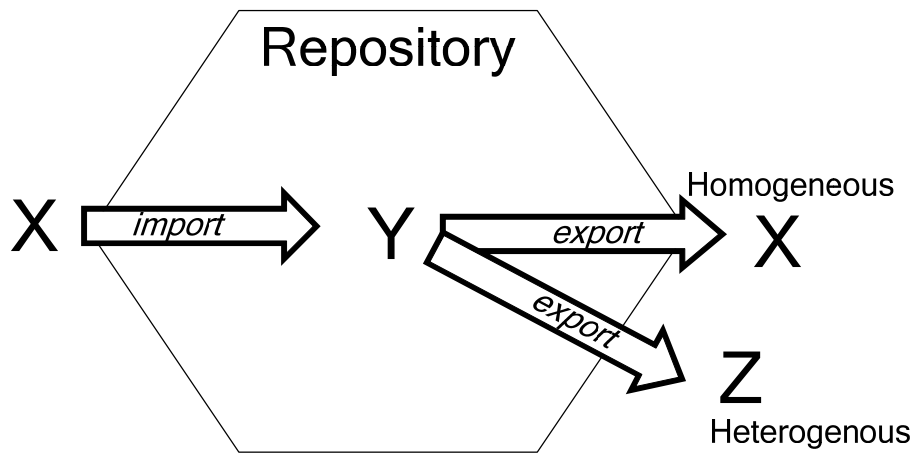
3084 Any combination of these five scenarios can be initiated by a team that is responsible for mapping an
3085 existing source into the CIM repository. Many other details must be addressed as the content of any of
3086 the sources changes or when the core or common model changes. When numerous existing sources are
3087 imported using all the import scenarios, we must consider the export side. Ignoring the technique
3088 partition, the possible export scenarios are as follows:

- 3089 • *From Recast Partition:* Create a recast mapping for a sub-partition in the recast partition to a
3090 standard group (that is, inverse of import 2). The desired method is to use the recast mapping to
3091 translate a standard group into a GDMO definition.
- 3092 • *From Recast Partition:* Create a domain mapping for a recast sub-partition to a known
3093 management model that is not the original source for the content that overlaps.
- 3094 • *From Domain Partition:* Create a recast mapping for the complete contents of the CIM schema
3095 to a selected technique (for MIF, this remapping results in a non-standard group).
- 3096 • *From Domain Partition:* Create a domain mapping for the contents of the CIM schema that
3097 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 homogeneous 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.

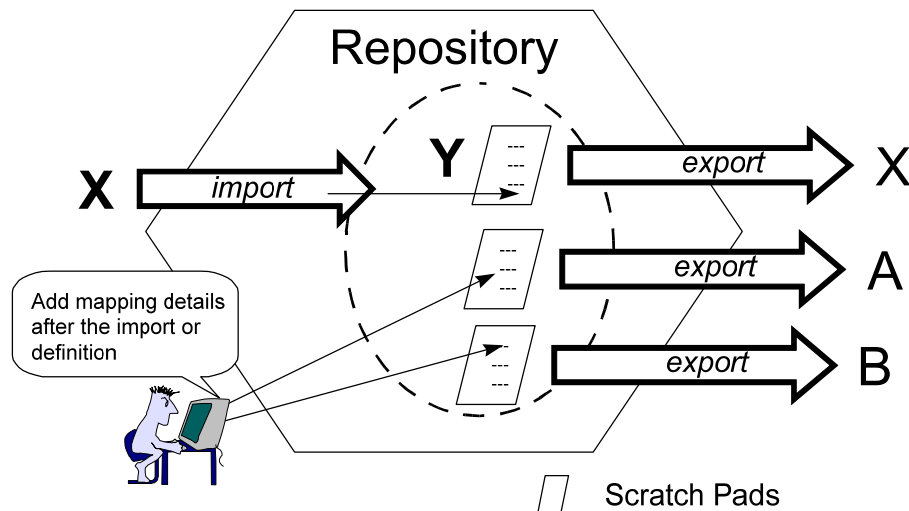


3112

3113

Figure 17 – Homogeneous and Heterogeneous Export

3114 The definition of the heterogeneous category is actually based on knowing how a schema is loaded into
 3115 the repository. To assist in understanding the export process, we can think of this process as using one of
 3116 multiple scratch pads. One scratch pad is created when the schema is loaded, and the others are added
 3117 to handle mappings to schema techniques other than the import source (Figure 18).



3118

3119

Figure 18 – Scratch Pads and Mapping

3120 Figure 18 shows how the scratch pads of qualifiers are used without factoring in the unique aspects of
 3121 each partition (technique, recast, applications) within the CIM repository. The next step is to consider
 3122 these partitions.

3123 For the technique partition, there is no need for a scratch pad because the CIM meta model is used to
 3124 describe the constructs in the source meta schema. Therefore, by definition, there is one homogeneous
 3125 mapping for each meta schema covered by the technique partition. These mappings create CIM objects
 3126 for the syntactic constructs of the schema and create associations for the ways they can be combined.
 3127 (For example, MIF groups include attributes.)

3128 For the recast partition, there are multiple scratch pads for each sub-partition because one is required for
 3129 each export target and there can be multiple mapping algorithms for each target. Multiple mapping
 3130 algorithms occur because part of creating a recast mapping involves mapping the constructs of the
 3131 source into CIM meta-model constructs. Therefore, for the MIF syntax, a mapping must be created for
 3132 component, group, attribute, and so on, into appropriate CIM meta-model constructs such as object,
 3133 association, property, and so on. These mappings can be arbitrary. For example, one decision to be
 3134 made is whether a group or a component maps into an object. Two different recast mapping algorithms
 3135 are possible: one that maps groups into objects with qualifiers that preserve the component, and one that
 3136 maps components into objects with qualifiers that preserve the group name for the properties. Therefore,
 3137 the scratch pads in the recast partition are organized by target technique and employed algorithm.

3138 For the domain partitions, there are two types of mappings:

- 3139 • A mapping similar to the recast partition in that part of the domain partition is mapped into the
 3140 syntax of another meta schema. These mappings can use the same qualifier scratch pads and
 3141 associated algorithms that are developed for the recast partition.
- 3142 • A mapping that facilitates documenting the content overlap between the domain partition and
 3143 another model (for example, software groups).

3144 These mappings cannot be determined in a generic way at import time; therefore, it is best to consider
3145 them in the context of exporting. The mapping uses filters to determine the overlaps and then performs
3146 the necessary conversions. The filtering can use qualifiers to indicate that a particular set of domain
3147 partition constructs maps into a combination of constructs in the target/source model. The conversions
3148 are documented in the repository using a complex set of qualifiers that capture how to write or insert the
3149 overlapped content into the target model. The mapping qualifiers for the domain partition are organized
3150 like the recasting partition for the syntax conversions, and there is a scratch pad for each model for
3151 documenting overlapping content.

3152 In summary, pick the partition, develop a mapping, and identify the qualifiers necessary to capture
3153 potentially lost information when mapping details are developed for a particular source. On the export
3154 side, the mapping algorithm verifies whether the content to be exported includes the necessary qualifiers
3155 for the logic to work.

ANNEX A (normative)

MOF Syntax Grammar Description

3156
3157
3158
3159

3160 This annex presents the grammar for MOF syntax. While the grammar is convenient for describing the
3161 MOF syntax clearly, the same MOF language can also be described by a different, LL(1)-parsable,
3162 grammar, which enables low-footprint implementations of MOF compilers. In addition, note these points:

- 3163 1) An empty property list is equivalent to "*" .
- 3164 2) All keywords are case-insensitive.
- 3165 3) The IDENTIFIER type is used for names of classes, properties, qualifiers, methods, and
3166 namespaces. The rules governing the naming of classes and properties are presented in
3167 ANNEX E.
- 3168 4) A string value may contain quote (") characters, if each is immediately preceded by a
3169 backslash (\) character.
- 3170 5) In the current release, the MOF BNF does not support initializing an array value to empty (an
3171 array with no elements). In the 3.0 version of this specification, the DMTF plans to extend the
3172 MOF BNF to support this functionality as follows:

3173 `arrayInitialize = "{" [arrayElementList] }`

3174 `arrayElementList = constantValue *("," constantValue)`

3175 To ensure interoperability with the V2.x implementations, the DMTF recommends that, where
3176 possible, the value of NULL rather than empty ({}) be used to represent the most common use
3177 cases. However, if this practice should cause confusion or other issues, implementations may
3178 use the syntax of the 3.0 version or higher to initialize an empty array.

3179 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 aliasIdentifier

aliasIdentifier      = "$" 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(unesaped)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 ";"

```

3180 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

```

3182
3183
3184
3185
3186 // =====
3187 //   NamedElement
3188 // =====
3189 [Version("2.3.0"), Description(
3190   "The Meta_NamedElement class represents the root class for the "
3191   "Metaschema. It has one property: Name, which is inherited by all the "
3192   "non-association classes in the Metaschema. Every metaconstruct is "
3193   "expressed as a descendent of the class Meta_Named Element.") ]
3194 class Meta_NamedElement
3195 {
3196   [Description (
3197     "The Name property indicates the name of the current Metaschema element. "
3198     "The following rules apply to the Name property, depending on the "
3199     "creation type of the object:<UL><LI>Fully-qualified class names, such "
3200     "as those prefixed by the schema name, are unique within the schema."
3201     "<LI>Fully-qualified association and indication names are unique within "
3202     "the schema (implied by the fact that association and indication classes "
3203     "are subtypes of Meta_Class). <LI>Implicitly-defined qualifier names are "
3204     "unique within the scope of the characterized object; that is, a named "
3205     "element may not have two characteristics with the same name."
3206     "<LI>Explicitly-defined qualifier names are unique within the defining "
3207     "schema. An implicitly-defined qualifier must agree in type, scope and "
3208     "flavor with any explicitly-defined qualifier of the same name."
3209     "<LI>Trigger names must be unique within the property, class or method "
3210     "to which the trigger applies. <LI>Method and property names must be "
3211     "unique within the domain class. A class can inherit more than one "
3212     "property or method with the same name. Property and method names can be "
3213     "qualified using the name of the declaring class. <LI>Reference names "
3214     "must be unique within the scope of their defining association class. "
3215     "Reference names obey the same rules as property names. </UL><B>Note:</B> "
3216     "Reference names are not required to be unique within the scope of the "
3217     "related class. Within such a scope, the reference provides the name of "
3218     "the class within the context defined by the association.") ]
3219   string Name;
3220 };
3221
3222 // =====
3223 //   QualifierFlavor
3224 // =====
3225 [Version("2.3.0"), Description (
3226   "The Meta_QualifierFlavor class encapsulates extra semantics attached "
3227   "to a qualifier such as the rules for transmission from superClass "
3228   "to subclass and whether or not the qualifier value may be translated "
3229   "into other languages") ]
3230 class Meta_QualifierFlavor:Meta_NamedElement
3231 {
3232 };
3233

```

```

3234 // =====
3235 //   Schema
3236 // =====
3237     [Version("2.3.0"), Description (
3238       "The Meta_Schema class represents a group of classes with a single owner."
3239       " Schemas are used for administration and class naming. Class names must "
3240       "be unique within their owning schemas.") ]
3241 class Meta_Schema:Meta_NamedElement
3242 {
3243 };
3244
3245 // =====
3246 //   Trigger
3247 // =====
3248     [Version("2.3.0"), Description (
3249       "A Trigger is a recognition of a state change (such as create, delete, "
3250       "update, or access) of a Class instance, and update or access of a "
3251       "Property.") ]
3252 class Meta_Trigger:Meta_NamedElement
3253 {
3254 };
3255
3256 // =====
3257 //   Qualifier
3258 // =====
3259     [Version("2.3.0"), Description (
3260       "The Meta_Qualifier class represents characteristics of named elements. "
3261       "For example, there are qualifiers that define the characteristics of a "
3262       "property or the key of a class. Qualifiers provide a mechanism that "
3263       "makes the Metaschema extensible in a limited and controlled fashion."
3264       "<P>It is possible to add new types of qualifiers by the introduction of "
3265       "a new qualifier name, thereby providing new types of metadata to "
3266       "processes that manage and manipulate classes, properties, and other "
3267       "elements of the Metaschema.") ]
3268 class Meta_Qualifier:Meta_NamedElement
3269 {
3270     [Description ("The Value property indicates the value of the qualifier.")]
3271     string Value;
3272 };
3273
3274 // =====
3275 //   Method
3276 // =====
3277     [Version( "2" ), Revision( "2" ), Description (
3278       "The Meta_Method class represents a declaration of a signature; that is, "
3279       "the method name, return type and parameters, and (in the case of a "
3280       "concrete class) may imply an implementation.") ]
3281 class Meta_Method:Meta_NamedElement
3282 {
3283 };
3284
3285 // =====
3286 //   Property
3287 // =====
3288     [Version( "2" ), Revision( "2" ), Description (
3289       "The Meta_Property class represents a value used to characterize "
3290       "instances of a class. A property can be thought of as a pair of Get and "
3291       "Set functions that, when applied to an object, return state and set "
3292       "state, respectively.") ]
3293 class Meta_Property:Meta_NamedElement
3294 {
3295 };
3296

```



```

3297 // =====
3298 // Reference
3299 // =====
3300 [Version( "2" ), Revision( "2" ), Description (
3301 "The Meta_Reference class represents (and defines) the role each object "
3302 "plays in an association. The reference represents the role name of a "
3303 "class in the context of an association, which supports the provision of "
3304 "multiple relationship instances for a given object. For example, a "
3305 "system can be related to many system components.") ]
3306 class Meta_Reference:Meta_Property
3307 {
3308 };
3309
3310 // =====
3311 // Class
3312 // =====
3313 [Version( "2" ), Revision( "2" ), Description (
3314 "The Meta_Class class is a collection of instances that support the same "
3315 "type; that is, the same properties and methods. Classes can be arranged "
3316 "in a generalization hierarchy that represents subtype relationships "
3317 "between classes. <P>The generalization hierarchy is a rooted, directed "
3318 "graph and does not support multiple inheritance. Classes can have "
3319 "methods, which represent the behavior relevant for that class. A Class "
3320 "may participate in associations by being the target of one of the "
3321 "references owned by the association.") ]
3322 class Meta_Class:Meta_NamedElement
3323 {
3324 };
3325
3326 // =====
3327 // Indication
3328 // =====
3329 [Version( "2" ), Revision( "2" ), Description (
3330 "The Meta_Indication class represents an object created as a result of a "
3331 "trigger. Because Indications are subtypes of Meta_Class, they can have "
3332 "properties and methods, and be arranged in a type hierarchy. ") ]
3333 class Meta_Indication:Meta_Class
3334 {
3335 };
3336
3337 // =====
3338 // Association
3339 // =====
3340 [Version( "2" ), Revision( "2" ), Description (
3341 "The Meta_Association class represents a class that contains two or more "
3342 "references and represents a relationship between two or more objects. "
3343 "Because of how associations are defined, it is possible to establish a "
3344 "relationship between classes without affecting any of the related "
3345 "classes.<P>For example, the addition of an association does not affect "
3346 "the interface of the related classes; associations have no other "
3347 "significance. Only associations can have references. Associations can "
3348 "be a subclass of a non-association class. Any subclass of "
3349 "Meta_Association is an association.") ]
3350 class Meta_Association:Meta_Class
3351 {
3352 };
3353

```

```

3354 // =====
3355 // Characteristics
3356 // =====
3357 [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
3358 "The Meta_Characteristics class relates a Meta_NamedElement to a "
3359 "qualifier that characterizes the named element. Meta_NamedElement may "
3360 "have zero or more characteristics.") ]
3361 class Meta_Characteristics
3362 {
3363     [Description (
3364 "The Characteristic reference represents the qualifier that "
3365 "characterizes the named element.") ]
3366     Meta_Qualifier REF Characteristic;
3367     [Aggregate, Description (
3368 "The Characterized reference represents the named element that is being "
3369 "characterized.") ]
3370     Meta_NamedElement REF Characterized;
3371 };
3372
3373 // =====
3374 // PropertyDomain
3375 // =====
3376 [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
3377 "The Meta_PropertyDomain class represents an association between a class "
3378 "and a property.<P>A property has only one domain: the class that owns "
3379 "the property. A property can have an override relationship with another "
3380 "property from a different class. The domain of the overridden property "
3381 "must be a supertype of the domain of the overriding property. The "
3382 "domain of a reference must be an association.") ]
3383 class Meta_PropertyDomain
3384 {
3385     [Description (
3386 "The Property reference represents the property that is owned by the "
3387 "class referenced by Domain.") ]
3388     Meta_Property REF Property;
3389     [Aggregate, Description (
3390 "The Domain reference represents the class that owns the property "
3391 "referenced by Property.") ]
3392     Meta_Class REF Domain;
3393 };
3394
3395 // =====
3396 // MethodDomain
3397 // =====
3398 [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
3399 "The Meta_MethodDomain class represents an association between a class "
3400 "and a method.<P>A method has only one domain: the class that owns the "
3401 "method, which can have an override relationship with another method "
3402 "from a different class. The domain of the overridden method must be a "
3403 "supertype of the domain of the overriding method. The signature of the "
3404 "method (that is, the name, parameters and return type) must be "
3405 "identical.") ]
3406 class Meta_MethodDomain
3407 {
3408     [Description (
3409 "The Method reference represents the method that is owned by the class "
3410 "referenced by Domain.") ]
3411     Meta_Method REF Method;
3412     [Aggregate, Description (
3413 "The Domain reference represents the class that owns the method "
3414 "referenced by Method.") ]
3415     Meta_Class REF Domain;
3416 };

```

```

3417
3418 // =====
3419 //   ReferenceRange
3420 // =====
3421     [Association, Version( "2" ), Revision( "2" ), Description (
3422       "The Meta_ReferenceRange class defines the type of the reference." ) ]
3423 class Meta_ReferenceRange
3424 {
3425     [Description (
3426       "The Reference reference represents the reference whose type is defined "
3427       "by Range." ) ]
3428     Meta_Reference REF Reference;
3429     [Description (
3430       "The Range reference represents the class that defines the type of "
3431       "reference." ) ]
3432     Meta_Class REF Range;
3433 };
3434
3435 // =====
3436 //   QualifiersFlavor
3437 // =====
3438     [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
3439       "The Meta_QualifiersFlavor class represents an association between a "
3440       "flavor and a qualifier." ) ]
3441 class Meta_QualifiersFlavor
3442 {
3443     [Description (
3444       "The Flavor reference represents the qualifier flavor to "
3445       "be applied to Qualifier." ) ]
3446     Meta_QualifierFlavor REF Flavor;
3447     [Aggregate, Description (
3448       "The Qualifier reference represents the qualifier to which "
3449       "Flavor applies." ) ]
3450     Meta_Qualifier REF Qualifier;
3451 };
3452
3453 // =====
3454 //   SubtypeSupertype
3455 // =====
3456     [Association, Version( "2" ), Revision( "2" ), Description (
3457       "The Meta_SubtypeSupertype class represents subtype/supertype "
3458       "relationships between classes arranged in a generalization hierarchy. "
3459       "This generalization hierarchy is a rooted, directed graph and does not "
3460       "support multiple inheritance." ) ]
3461 class Meta_SubtypeSupertype
3462 {
3463     [Description (
3464       "The SuperClass reference represents the class that is hierarchically "
3465       "immediately above the class referenced by SubClass." ) ]
3466     Meta_Class REF SuperClass;
3467     [Description (
3468       "The SubClass reference represents the class that is the immediate "
3469       "descendent of the class referenced by SuperClass." ) ]
3470     Meta_Class REF SubClass;
3471 };
3472

```

```

3473 // =====
3474 //   PropertyOverride
3475 // =====
3476 [Association, Version( "2" ), Revision( "2" ), Description (
3477   "The Meta_PropertyOverride class represents an association between two "
3478   "properties where one overrides the other.<P>Properties have reflexive "
3479   "associations that represent property overriding. A property can "
3480   "override an inherited property, which implies that any access to the "
3481   "inherited property will result in the invocation of the implementation "
3482   "of the overriding property. A Property can have an override "
3483   "relationship with another property from a different class.<P>The domain "
3484   "of the overridden property must be a supertype of the domain of the "
3485   "overriding property. The class referenced by the Meta_ReferenceRange "
3486   "association of an overriding reference must be the same as, or a "
3487   "subtype of, the class referenced by the Meta_ReferenceRange "
3488   "associations of the reference being overridden.") ]
3489 class Meta_PropertyOverride
3490 {
3491     [Description (
3492     "The OverridingProperty reference represents the property that overrides "
3493     "the property referenced by OverriddenProperty.") ]
3494     Meta_Property REF OverridingProperty;
3495     [Description (
3496     "The OverriddenProperty reference represents the property that is "
3497     "overridden by the property reference by OverridingProperty.") ]
3498     Meta_Property REF OverriddenProperty;
3499 };
3500
3501 // =====
3502 //   MethodOverride
3503 // =====
3504 [Association, Version( "2" ), Revision( "2" ), Description (
3505   "The Meta_MethodOverride class represents an association between two "
3506   "methods, where one overrides the other. Methods have reflexive "
3507   "associations that represent method overriding. A method can override an "
3508   "inherited method, which implies that any access to the inherited method "
3509   "will result in the invocation of the implementation of the overriding "
3510   "method.") ]
3511 class Meta_MethodOverride
3512 {
3513     [Description (
3514     "The OverridingMethod reference represents the method that overrides the "
3515     "method referenced by OverriddenMethod.") ]
3516     Meta_Method REF OverridingMethod;
3517     [Description (
3518     "The OverriddenMethod reference represents the method that is overridden "
3519     "by the method reference by OverridingMethod.") ]
3520     Meta_Method REF OverriddenMethod;
3521 };
3522
3523 // =====
3524 //   ElementSchema
3525 // =====
3526 [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
3527   "The Meta_ElementSchema class represents the elements (typically classes "
3528   "and qualifiers) that make up a schema.") ]
3529 class Meta_ElementSchema
3530 {
3531     [Description (
3532     "The Element reference represents the named element that belongs to the "
3533     "schema referenced by Schema.") ]
3534     Meta_NamedElement REF Element;
3535     [Aggregate, Description (

```

```
3536         "The Schema reference represents the schema to which the named element "  
3537         "referenced by Element belongs.") ]  
3538     Meta_Schema REF Schema;  
3539 };
```

ANNEX C (normative)

Units

3540
3541
3542
3543

3544 C.1 Programmatic Units

3545 This annex defines the concept and syntax of a programmatic unit, which is an expression of a unit of
3546 measure for programmatic access. It makes it easy to recognize the base units of which the actual unit is
3547 made, as well as any numerical multipliers. Programmatic units are used as a value for the PUnit qualifier
3548 and also as a value for any (string typed) CIM elements that represent units. The Boolean IsPUnit
3549 qualifier is used to declare that a string typed element follows the syntax for programmatic units.

3550 Programmatic units must be processed case-sensitively and white-space-sensitively.

3551 As defined in the Augmented BNF (ABNF) syntax, the programmatic unit consists of a base unit that is
3552 optionally followed by other base units that are each either multiplied or divided into the first base unit.
3553 Furthermore, two optional multipliers can be applied. The first is simply a scalar, and the second is an
3554 exponential number consisting of a base and an exponent. The optional multipliers enable the
3555 specification of common derived units of measure in terms of the allowed base units. Note that the base
3556 units defined in this subclause include a superset of the SI base units. When a unit is the empty string,
3557 the value has no unit; that is, it is dimensionless. The multipliers must be understood as part of the
3558 definition of the derived unit; that is, scale prefixes of units are replaced with their numerical value. For
3559 example, "kilometer" is represented as "meter * 1000", replacing the "kilo" scale prefix with the numerical
3560 factor 1000.

3561 A string representing a programmatic unit must follow the production "programmatic-unit" in the syntax
3562 defined in this annex. This syntax supports any type of unit, including SI units, United States units, and
3563 any other standard or non-standard units. The syntax definition here uses [ABNF](#) with the following
3564 exceptions:

- 3565 • Rules separated by a bar (|) represent choices (instead of using a forward slash (/) as
3566 defined in ABNF).
- 3567 • Any characters must be processed case sensitively instead of case-insensitively, as defined in
3568 ABNF.

3569 ABNF defines the items in the syntax as assembled without inserted white space. Therefore, the syntax
3570 explicitly specifies any white space. The ABNF syntax is defined as follows:

3571 programmatic-unit = ("" | base-unit *([WS] multiplied-base-unit) *([WS] divided-base-unit) [[WS]
3572 modifier1] [[WS] modifier2])

3573 multiplied-base-unit = "*" [WS] base-unit

3574 divided-base-unit = "/" [WS] base-unit

3575 modifier1 = operator [WS] number

3576 modifier2 = operator [WS] base [WS] "^" [WS] exponent

3577 operator = "*" | "/"

3578 number = ["+" | "-"] positive-number

3579 base = positive-whole-number

3580 exponent = ["+" | "-"] positive-whole-number

3581 positive-whole-number = NON-ZERO-DIGIT *(DIGIT)

3582 positive-number = positive-whole-number | ((positive-whole-number | ZERO) "." *(DIGIT))

3583 base-unit = simple-name | decibel-base-unit

3584 simple-name = FIRST-UNIT-CHAR *([S] UNIT-CHAR)

3585 decibel-base-unit = "decibel" [[S] "(" [S] simple-name [S] ")"]

3586 FIRST-UNIT-CHAR = ("A"..."Z" | "a"..."z" | "_" | U+0080...U+FFEF)

3587 UNIT-CHAR = (FIRST-UNIT-CHAR | "0"..."9" | "-")

3588 ZERO = "0"

3589 NON-ZERO-DIGIT = ("1"..."9")

3590 DIGIT = ZERO | NON-ZERO-DIGIT

3591 WS = (S | TAB | NL)

3592 S = U+0020

3593 TAB = U+0009

3594 NL = U+000A

3595 Unicode characters used in the syntax:

3596 U+0009 = "\t" (tab)

3597 U+000A = "\n" (newline)

3598 U+0020 = " " (space)

3599 U+0080...U+FFEF = (other Unicode characters)

3600 For example, a speedometer may be modeled so that the unit of measure is kilometers per hour. It is
 3601 necessary to express the derived unit of measure "kilometers per hour" in terms of the allowed base units
 3602 "meter" and "second". One kilometer per hour is equivalent to

3603 1000 meters per 3600 seconds

3604 or

3605 one meter / second / 3.6

3606 so the programmatic unit for "kilometers per hour" is expressed as: "meter / second / 3.6", using the
 3607 syntax defined here.

3608 Other examples are as follows:

3609 "meter * meter * 10⁻⁶" → square millimeters

3610 "byte * 2¹⁰" → kBytes as used for memory ("kibobyte")

3611 "byte * 10³" → kBytes as used for storage ("kilobyte")

3612 "dataword * 4" → QuadWords

3613 "decibel(m) * -1" → -dBm

3614 "second * 250 * 10⁻⁹" → 250 nanoseconds

3615 "foot * foot * foot / minute" → cubic feet per minute, CFM

3616 "revolution / minute" → revolutions per minute, RPM

3617 "pound / inch / inch" → pounds per square inch, PSI

3618 "foot * pound" → foot-pounds

3619 In the "PU Base Unit" column, Table C-1 defines the allowed values for the production "base-unit" in
 3620 the syntax, as well as the empty string indicating no unit. The "Symbol" column recommends a
 3621 symbol to be used in a human interface. The "Calculation" column relates units to other units. The
 3622 "Quantity" column lists the physical quantity measured by the unit.

3623 The base units in Table C-1 consist of the SI base units and the SI derived units amended by other
 3624 commonly used units. Note that "SI" is the international abbreviation for the International System of Units
 3625 (French: "Système International d'Unites"), defined in [ISO 1000:1992](#). Also, [ISO 1000:1992](#) defines the
 3626 notational conventions for units, which are used in Table C-1.

3627 **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/s ²	Acceleration
degree celsius	°C	1 °C = 1 K (diff)	Thermodynamic temperature
degree fahrenheit	°F	1 °F = 5/9 K (diff)	Thermodynamic temperature
kelvin	K	SI base unit	Thermodynamic temperature, color temperature
candela	cd	SI base unit	Luminous intensity
lumen	lm	1 lm = 1 cd·sr	Luminous flux
nit	nit	1 nit = 1 cd/m ²	Luminance
lux	lx	1 lx = 1 lm/m ²	Illuminance
newton	N	1 N = 1 kg·m/s ²	Force
pascal	Pa	1 Pa = 1 N/m ²	Pressure
bar	bar	1 bar = 100000 Pa	Pressure
decibel(A)	dB(A)	1 dB(A) = 20 lg (p/p ₀)	Loudness of sound, relative to reference sound pressure level of p ₀ = 20 μPa in gases, using frequency weight curve (A)
decibel(C)	dB(C)	1 dB(C) = 20 · lg (p/p ₀)	Loudness of sound, relative to reference sound pressure level of p ₀ = 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/P ₀)	Power, relative to reference power of P ₀ = 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

3628 c.2 Value for Units Qualifier

3629 **Deprecated:** The Units qualifier has been used both for programmatic access and for displaying a unit.
 3630 Because it does not satisfy the full needs of either of these uses, the Units qualifier is deprecated. The
 3631 PUnit qualifier should be used instead for programmatic access. For displaying a unit, the client
 3632 application should construct the string to be displayed from the PUnit qualifier using the conventions of
 3633 the client application.

3634 The UNITS qualifier specifies the unit of measure in which the qualified property, method return value, or
 3635 method parameter is expressed. For example, a Size property might have Units (Bytes). The complete
 3636 set of DMTF-defined values for the Units qualifier is as follows:

- 3637 • Bits, KiloBits, MegaBits, GigaBits
- 3638 • < Bits, KiloBits, MegaBits, GigaBits> per Second
- 3639 • Bytes, KiloBytes, MegaBytes, GigaBytes, Words, DoubleWords, QuadWords
- 3640 • Degrees C, Tenths of Degrees C, Hundredths of Degrees C, Degrees F, Tenths of Degrees F,
 3641 Hundredths of Degrees F, Degrees K, Tenths of Degrees K, Hundredths of Degrees K, Color
 3642 Temperature
- 3643 • Volts, MilliVolts, Tenths of MilliVolts, Amps, MilliAmps, Tenths of MilliAmps, Watts,
 3644 MilliWattHours
- 3645 • Joules, Coulombs, Newtons
- 3646 • Lumen, Lux, Candelas
- 3647 • Pounds, Pounds per Square Inch
- 3648 • Cycles, Revolutions, Revolutions per Minute, Revolutions per Second
- 3649 • Minutes, Seconds, Tenths of Seconds, Hundredths of Seconds, MicroSeconds, MilliSeconds,
 3650 NanoSeconds
- 3651 • Hours, Days, Weeks
- 3652 • Hertz, MegaHertz
- 3653 • Pixels, Pixels per Inch
- 3654 • Counts per Inch
- 3655 • Percent, Tenths of Percent, Hundredths of Percent, Thousandths
- 3656 • Meters, Centimeters, Millimeters, Cubic Meters, Cubic Centimeters, Cubic Millimeters
- 3657 • Inches, Feet, Cubic Inches, Cubic Feet, Ounces, Liters, Fluid Ounces

- 3658 • Radians, Steradians, Degrees
- 3659 • Gravities, Pounds, Foot-Pounds
- 3660 • Gauss, Gilberts, Henrys, MilliHenrys, Farads, MilliFarads, MicroFarads, PicoFarads
- 3661 • Ohms, Siemens
- 3662 • Moles, Becquerels, Parts per Million
- 3663 • Decibels, Tenths of Decibels
- 3664 • Grays, Sieverts
- 3665 • MilliWatts
- 3666 • DBm
- 3667 • <Bytes, KiloBytes, MegaBytes, GigaBytes> per Second
- 3668 • BTU per Hour
- 3669 • PCI clock cycles
- 3670 • <Numeric value> <Minutes, Seconds, Tenths of Seconds, Hundreths of Seconds,
3671 MicroSeconds, MilliSeconds, Nanoseconds>
- 3672 • Us³
- 3673 • Amps at <Numeric Value> Volts
- 3674 • Clock Ticks
- 3675 • Packets, per Thousand Packets

³ Standard Rack Measurement equal to 1.75 inches.

ANNEX D (informative)

UML Notation

3676
3677
3678
3679

3680 The CIM meta-schema notation is directly based on the notation used in Unified Modeling Language
3681 (UML). There are distinct symbols for all the major constructs in the schema except qualifiers (as opposed
3682 to properties, which are directly represented in the diagrams).

3683 In UML, a class is represented by a rectangle. The class name either stands alone in the rectangle or is in
3684 the uppermost segment of the rectangle. If present, the segment below the segment with the name
3685 contains the properties of the class. If present, a third region contains methods.

3686 A line decorated with a triangle indicates an inheritance relationship; the lower rectangle represents a
3687 subtype of the upper rectangle. The triangle points to the superclass.

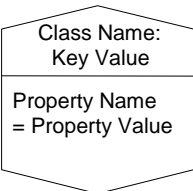
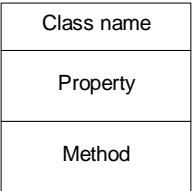
3688 Other solid lines represent relationships. The cardinality of the references on either side of the
3689 relationship is indicated by a decoration on either end. The following character combinations are
3690 commonly used:

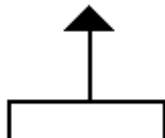
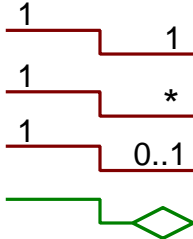
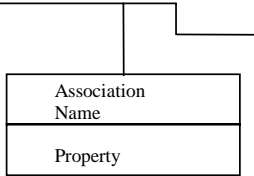
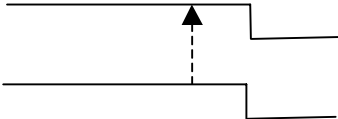
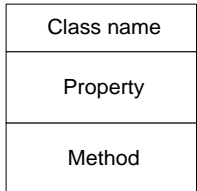
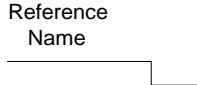
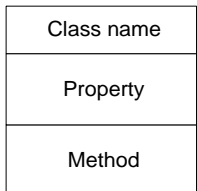
- 3691 • "1" indicates a single-valued, required reference
- 3692 • "0..1" indicates an optional single-valued reference
- 3693 • "*" indicates an optional many-valued reference (as does "0..*")
- 3694 • "1..*" indicates a required many-valued reference

3695 A line connected to a rectangle by a dotted line represents a subclass relationship between two
3696 associations. The diagramming notation and its interpretation are summarized in Table D-1.

3697

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	

3698

ANNEX E (normative)

Unicode Usage

3699
3700
3701
3702

3703 All punctuation symbols associated with object path or MOF syntax occur within the Basic Latin range
3704 U+0000 to U+007F. These symbols include normal punctuators, such as slashes, colons, commas, and
3705 so on. No important syntactic punctuation character occurs outside of this range.

3706 All characters above U+007F are treated as parts of names, even though there are several reserved
3707 characters such as U+2028 and U+2029, which are logically white space. Therefore, all namespace,
3708 class, and property names are identifiers composed as follows:

- 3709 • Initial identifier characters must be in set S1, where S1 = {U+005F, U+0041...U+005A,
3710 U+0061...U+007A, U+0080...U+FFEF} (This includes alphabetic characters and the
3711 underscore.)
- 3712 • All following characters must be in set S2 where S2 = S1 union {U+0030...U+0039} (This
3713 includes alphabetic characters, Arabic numerals 0 through 9, and the underscore.)

3714 Note that the Unicode specials range (U+FFFF0...U+FFFF) are not legal for identifiers. While the preceding
3715 sub-range of U+0080...U+FFEF includes many diacritical characters that would not be useful in an
3716 identifier, as well as the Unicode reserved sub-range that is not allocated, it seems advisable for simplicity
3717 of parsers simply to treat this entire sub-range as legal for identifiers.

3718 Refer to [RFC2279](#) for an example of a Universal Transformation Format with specific characteristics for
3719 dealing with multi-octet characters on an application-specific basis.

3720 E.1 MOF Text

3721 MOF files using Unicode must contain a signature as the first two bytes of the text file, either U+FFFE or
3722 U+FEFF, depending on the byte ordering of the text file (as suggested in Section 2.4 of the [ISO/IEC](#)
3723 [10646:2003](#)). U+FFFE is little endian.

3724 All MOF keywords and punctuation symbols are as described in the MOF syntax document and are not
3725 locale-specific. They are composed of characters falling in the range U+0000...U+007F, regardless of the
3726 locale of origin for the MOF or its identifiers.

3727 E.2 Quoted Strings

3728 In all cases where non-identifier string values are required, delimiters must surround them. The supported
3729 delimiter for strings is U+0027. When a quoted string is started using the delimiter, the same delimiter,
3730 U+0027, is used to terminate it. In addition, the digraph U+005C ("\\") followed by U+0027 "" constitutes
3731 an embedded quotation mark, not a termination of the quoted string. The characters permitted within
3732 these quotation mark delimiters may fall within the range U+0001 through U+FFEF.

ANNEX F (informative)

Guidelines

3733
3734
3735
3736

3737 The following are guidelines for modeling:

- 3738 • Method descriptions are recommended and must, at a minimum, indicate the method's side
3739 effects (pre- and post-conditions).
- 3740 • Associations must not be declared as subtypes of classes that are not associations.
- 3741 • Leading underscores in identifiers are to be discouraged and not used at all in the standard
3742 schemas.
- 3743 • It is generally recommended that class names not be reused as part of property or method
3744 names. Property and method names are already unique within their defining class.
- 3745 • To enable information sharing among different CIM implementations, the MaxLen qualifier
3746 should be used to specify the maximum length of string properties. This qualifier must *always*
3747 be present for string properties used as keys.
- 3748 • A class with no Abstract qualifier must define, or inherit, key properties.

3749 F.1 Mapping of Octet Strings

3750 Most management models, including SNMP and DMI, support octet strings as data types. The octet string
3751 data type represents arbitrary numeric or textual data that is stored as an indexed byte array of unlimited
3752 but fixed size. Typically, the first n bytes indicate the actual string length. Because some environments
3753 reserve only the first byte, they do not support octet strings larger than 255 bytes.

3754 In the current release, CIM does not support octet strings as a separate data type. To map a single octet
3755 string (that is, an octet of binary data), the equivalent CIM property should be defined as an array of
3756 unsigned 8-bit integers (uint8). The first four bytes of the array contain the length of the octet data: byte 0
3757 is the most significant byte of the length, and byte 3 is the least significant byte. The octet data starts at
3758 byte 4. The OctetString qualifier may be used to indicate that the uint8 array conforms to this encoding.

3759 Arrays of uint8 arrays are not supported. Therefore, to map an array of octet strings, a textual convention
3760 encoding the binary information as hexadecimal digit characters (such as 0x<<0-9,A-F><0-9,A-F>>*) is
3761 used for each octet string in the array. The number of octets in the octet string is encoded in the first 8
3762 hexadecimal digits of the string with the most significant digits in the left-most characters of the string. The
3763 length count octets are included in the length count. For example, "0x00000004" is the encoding of a 0-
3764 length octet string.

3765 The OctetString qualifier qualifies the string array.

3766 EXAMPLE: Example use of the OctetString qualifier on a property is as follows:

```
3767 [Description ("An octet string"), Octetstring]  
3768 uint8 Foo[];  
3769 [Description ("An array of octet strings"), Octetstring]  
3770 String Bar[];
```


3771 **F.2 SQL Reserved Words**

3772 Avoid using SQL reserved words in class and property names. This restriction particularly applies to
 3773 property names because class names are prefixed by the schema name, making a clash with a reserved
 3774 word unlikely. The current set of SQL reserved words is as follows:

3775 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	

3776 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
ASYNCH	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

3783 Use of the EmbeddedObject and EmbeddedInstance qualifiers is motivated by the need to include the
3784 data of a specific instance in an indication (event notification) or to capture the contents of an instance at
3785 a point in time (for example, to include the CIM_DiagnosticSetting properties that dictate a particular
3786 CIM_DiagnosticResult in the Result object).

3787 Therefore, the next major version of the CIM Specification is expected to include a separate data type for
3788 directly representing instances (or snapshots of instances). Until then, the EmbeddedObject and
3789 EmbeddedInstance qualifiers can be used to achieve an approximately equivalent effect. They permit a
3790 CIM object manager (or other entity) to simulate embedded instances or classes by encoding them as
3791 strings when they are presented externally. Clients that do not handle embedded objects may treat
3792 properties with this qualifier just like any other string-valued property. Clients that do want to realize the
3793 capability of embedded objects can extract the embedded object information by decoding the presented
3794 string value.

3795 To reduce the parsing burden, the encoding that represents the embedded object in the string value
3796 depends on the protocol or representation used for transmitting the containing instance. This dependency
3797 makes the string value appear to vary according to the circumstances in which it is observed. This is an
3798 acknowledged weakness of using a qualifier instead of a new data type.

3799 This document defines the encoding of embedded objects for the MOF representation and for the CIM-
3800 XML protocol. When other protocols or representations are used to communicate with embedded object-
3801 aware consumers of CIM data, they must include particulars on the encoding for the values of string-
3802 typed elements qualified with EmbeddedObject or EmbeddedInstance.

3803 G.1 Encoding for MOF

3804 When the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance are
3805 rendered in MOF, the embedded object must be encoded into string form using the MOF syntax for the
3806 instanceDeclaration nonterminal in embedded instances or for the classDeclaration, assocDeclaration, or
3807 indicDeclaration nonterminals, as appropriate in embedded classes (see ANNEX A).

3808 EXAMPLE:

```
3809 Instance of CIM_InstCreation {
3810     EventTime = "20000208165854.457000-360";
3811     SourceInstance =
3812         "Instance of CIM_FAN {"
3813         "DeviceID = \"Fan 1\";"
3814         "Status = \"Degraded\";"
3815         "};";
3816 };
3817 Instance of CIM_ClassCreation {
3818     EventTime = "20031120165854.457000-360";
3819     ClassDefinition =
3820         "class CIM_Fan : CIM_CoolingDevice {"
3821         " boolean VariableSpeed;"
3822         " [Units (\"Revolutions per Minute\") ]"
3823         " uint64 DesiredSpeed;"
3824         "};";
3825 };
```

3826 G.2 Encoding for CIM-XML

3827 When the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance are
3828 rendered in CIM-XML, the embedded object must be encoded into string form as either an INSTANCE
3829 element (for instances) or a CLASS element (for classes), as defined in the DMTF [DSP0200](#), and
3830 [DSP0201](#).

ANNEX H (informative)

Schema Errata

3831
3832
3833
3834

3835 Based on the concepts and constructs in this specification, the CIM schema is expected to evolve for the
3836 following reasons:

- 3837 • To add new classes, associations, qualifiers, properties and/or methods. This task is addressed
3838 in 5.3.
- 3839 • To correct errors in the Final Release versions of the schema. This task fixes errata in the CIM
3840 schemas after their final release.
- 3841 • To deprecate and update the model by labeling classes, associations, qualifiers, and so on as
3842 "not recommended for future development" and replacing them with new constructs. This task is
3843 addressed by the Deprecated qualifier described in 5.5.2.11.

3844 Examples of errata to correct in CIM schemas are as follows:

- 3845 • Incorrectly or incompletely defined keys (an array defined as a key property, or incompletely
3846 specified propagated keys)
- 3847 • Invalid subclassing, such as subclassing an optional association from a weak relationship (that
3848 is, a mandatory association), subclassing a nonassociation class from an association, or
3849 subclassing an association but having different reference names that result in three or more
3850 references on an association
- 3851 • Class references reversed as defined by an association's roles (antecedent/dependent
3852 references reversed)
- 3853 • Use of SQL reserved words as property names
- 3854 • Violation of semantics, such as Missing Min(1) on a Weak relationship, contradicting that a
3855 Weak relationship is mandatory

3856 Errata are a serious matter because the schema should be correct, but the needs of existing
3857 implementations must be taken into account. Therefore, the DMTF has defined the following process (in
3858 addition to the normal release process) with respect to any schema errata:

- 3859 a) Any error should promptly be reported to the Technical Committee (technical@dmf.org) for
3860 review. Suggestions for correcting the error should also be made, if possible.
- 3861 b) The Technical Committee documents its findings in an email message to the submitter within
3862 21 days. These findings report the Committee's decision about whether the submission is a
3863 valid erratum, the reasoning behind the decision, the recommended strategy to correct the
3864 error, and whether backward compatibility is possible.
- 3865 c) If the error is valid, an email message is sent (with the reply to the submitter) to all DMTF
3866 members (members@dmf.org). The message highlights the error, the findings of the Technical
3867 Committee, and the strategy to correct the error. In addition, the committee indicates the
3868 affected versions of the schema (that is, only the latest or all schemas after a specific version).
- 3869 d) All members are invited to respond to the Technical Committee within 30 days regarding the
3870 impact of the correction strategy on their implementations. The effects should be explained as
3871 thoroughly as possible, as well as alternate strategies to correct the error.

- 3872 e) If one or more members are affected, then the Technical Committee evaluates all proposed
3873 alternate correction strategies. It chooses one of the following three options:
- 3874 – To stay with the correction strategy proposed in b)
 - 3875 – To move to one of the proposed alternate strategies
 - 3876 – To define a new correction strategy based on the evaluation of member impacts
- 3877 f) If an alternate strategy is proposed in Item e), the Technical Committee may decide to reenter
3878 the errata process, resuming with Item c) and send an email message to all DMTF members
3879 about the alternate correction strategy. However, if the Technical Committee believes that
3880 further comment will not raise any new issues, then the outcome of Item e) is declared to be
3881 final.
- 3882 g) If a final strategy is decided, this strategy is implemented through a Change Request to the
3883 affected schema(s). The Technical Committee writes and issues the Change Request. Affected
3884 models and MOF are updated, and their introductory comment section is flagged to indicate that
3885 a correction has been applied.

ANNEX I (informative)

Ambiguous Property and Method Names

3886
3887
3888
3889

3890 In 5.1, item 21)-e) explicitly allows a subclass to define a property that may have the same name as a
3891 property defined by a superclass and for that new property not to override the superclass property. The
3892 subclass may override the superclass property by attaching an Override qualifier; this situation is well-
3893 behaved and is not part of the problem under discussion.

3894 Similarly, a subclass may define a method with the same name as a method defined by a superclass
3895 without overriding the superclass method. This annex refers only to properties, but it is to be understood
3896 that the issues regarding methods are essentially the same. For any statement about properties, a similar
3897 statement about methods can be inferred.

3898 This same-name capability allows one group (the DMTF, in particular) to enhance or extend the
3899 superclass in a minor schema change without to coordinate with, or even to know about, the development
3900 of the subclass in another schema by another group. That is, a subclass defined in one version of the
3901 superclass should not become invalid if a subsequent version of the superclass introduces a new
3902 property with the same name as a property defined on the subclass. Any other use of the same-name
3903 capability is strongly discouraged, and additional constraints on allowable cases may well be added in
3904 future versions of CIM.

3905 It is natural for CIM applications to be written under the assumption that property names alone suffice to
3906 identify properties uniquely. However, such applications risk failure if they refer to properties from a
3907 subclass whose superclass has been modified to include a new property with the same name as a
3908 previously-existing property defined by the subclass. For example, consider the following:

```
3909     [abstract]
3910     class CIM_Superclass
3911     {
3912     };
3913
3914     class VENDOR_Subclass
3915     {
3916         string      Foo;
3917     };
```

3918 If there is just one instance of VENDOR_Subclass, a call to enumerateInstances("VENDOR_Subclass")
3919 might produce the following XML result from the CIMOM if it did not bother to ask for CLASSORIGIN
3920 information:

```
3921     <INSTANCE CLASSNAME="VENDOR_Subclass">
3922         <PROPERTY NAME="Foo" TYPE="string">
3923             <VALUE>Hello, my name is Foo</VALUE>
3924         </PROPERTY>
3925     </INSTANCE>
```

3926

3927 If the definition of CIM_Superclass changes to:

```
3928     [abstract]
3929     class CIM_Superclass
3930     {
3931         string foo = "You lose!";
3932     };
```

3933 then the enumerateInstances call might return the following:

```
3934     <INSTANCE>
3935         <PROPERTY NAME="Foo" TYPE="string">
3936             <VALUE>You lose!</VALUE>
3937         </PROPERTY>
3938         <PROPERTY NAME="Foo" TYPE="string">
3939             <VALUE>Hello, my name is Foo</VALUE>
3940         </PROPERTY>
3941     </INSTANCE>
```

3942 If the client application attempts to retrieve the 'foo' property, the value it obtains (if it does not experience
3943 an error) depends on the implementation.

3944 Although a class may define a property with the same name as an inherited property, it may not define
3945 two (or more) properties with the same name. Therefore, the combination of defining class plus property
3946 name uniquely identifies a property. (Most CIM operations that return instances have a flag controlling
3947 whether to include the originClass for each property. For example, in [DSP0200](#), see the clause on
3948 enumerateInstances; in [DSP0201](#), see the clause on ClassOrigin.)

3949 However, the use of class-plus-property-name for identifying properties makes an application vulnerable
3950 to failure if a property is promoted to a superclass in a subsequent schema release. For example,
3951 consider the following:

```
3952     class CIM_Top
3953     {
3954     };
3955
3956     class CIM_Middle : CIM_Top
3957     {
3958         uint32    foo;
3959     };
3960
3961     class VENDOR_Bottom : CIM_Middle
3962     {
3963         string    foo;
3964     };
```

3965 An application that identifies the uint32 property as "the property named 'foo' defined by CIM_Middle" no
3966 longer works if a subsequent release of the CIM schema changes the hierarchy as follows:

```
3967     class CIM_Top
3968     {
3969         uint32    foo;
3970     };
3971
3972     class CIM_Middle : CIM_Top
3973     {
3974     };
```



```
3975
3976     class VENDOR_Bottom : CIM_Middle
3977     {
3978         string     foo;
3979     };
```

3980 Strictly speaking, there is no longer a "property named 'foo' defined by CIM_Middle"; it is now defined by
3981 CIM_Top and merely inherited by CIM_Middle, just as it is inherited by VENDOR_Bottom. An instance of
3982 VENDOR_Bottom returned in XML from a CIMOM might look like this:

```
3983     <INSTANCE CLASSNAME="VENDOR_Bottom">
3984         <PROPERTY NAME="Foo" TYPE="string" CLASSORIGIN="VENDOR_Bottom">
3985             <VALUE>Hello, my name is Foo!</VALUE>
3986         </PROPERTY>
3987         <PROPERTY NAME="Foo" TYPE="uint32" CLASSORIGIN="CIM_Top">
3988             <VALUE>47</VALUE>
3989         </PROPERTY>
3990     </INSTANCE>
```

3991 A client application looking for a PROPERTY element with NAME="Foo" and
3992 CLASSORIGIN="CIM_Middle" fails with this XML fragment.

3993 Although CIM_Middle no longer defines a 'foo' property directly in this example, we intuit that we should
3994 be able to point to the CIM_Middle class and locate the 'foo' property that is defined in its nearest
3995 superclass. Generally, the application must be prepared to perform this search, separately obtaining
3996 information, when necessary, about the (current) class hierarchy and implementing an algorithm to select
3997 the appropriate property information from the instance information returned from a server operation.

3998 Although it is technically allowed, schema writers should not introduce properties that cause name
3999 collisions within the schema, and they are strongly discouraged from introducing properties with names
4000 known to conflict with property names of any subclass or superclass in another schema.

ANNEX J (informative)

OCL Considerations

4001
4002
4003
4004

4005 The Object Constraint Language (OCL) is a formal language to describe expressions on models. It is
4006 defined by the Open Management Group (OMG) in the [Object Constraint Language Specification](#), which
4007 describes OCL as follows:

4008 "OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without
4009 side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change
4010 anything in the model. This means that the state of the system will never change because of the
4011 evaluation of an OCL expression, even though an OCL expression can be used to specify a state
4012 change (e.g., in a post-condition).

4013 OCL is not a programming language; therefore, it is not possible to write program logic or flow
4014 control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because
4015 OCL is a modeling language in the first place, OCL expressions are not by definition directly
4016 executable.

4017 OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL
4018 expression must conform to the type conformance rules of the language. For example, you cannot
4019 compare an Integer with a String. Each Classifier defined within a UML model represents a distinct
4020 OCL type. In addition, OCL includes a set of supplementary predefined types. These are described
4021 in Chapter 11 ("The OCL Standard Library").

4022 As a specification language, all implementation issues are out of scope and cannot be expressed in
4023 OCL. The evaluation of an OCL expression is instantaneous. This means that the states of objects in
4024 a model cannot change during evaluation."

4025 For a particular CIM class, more than one CIM association referencing that class with one reference can
4026 define the same name for the opposite reference. OCL allows navigation from an instance of such a class
4027 to the instances at the other end of an association using the name of the opposite association end (that
4028 is, a CIM reference). However, in the case discussed, that name is not unique. For OCL statements to
4029 tolerate the future addition of associations that create such ambiguity, OCL navigation from an instance to
4030 any associated instances should first navigate to the association class and from there to the associated
4031 class, as described in the [Object Constraint Language Specification](#) in sections 7.5.4 "Navigation to
4032 Association Classes" and 7.5.5 "Navigation from Association Classes". Note that OCL requires the first
4033 letter of the association class name to be lowercase when used for navigating to it. For example,
4034 CIM_Dependency becomes cim_Dependency.

4035 EXAMPLE:

```
4036 [ClassConstraint {  
4037   "inv i1: self.p1 = self.a12.r.p2"}]  
4038 // Using a12 is required to disambiguate end name r  
4039 class C1 {  
4040   string p1;  
4041 };  
4042 [ClassConstraint {  
4043   "inv i2: self.p2 = self.a12.x.p1", // Using a12 is recommended  
4044   "inv i3: self.p2 = self.x.p1"}] // Works, but not recommended  
4045 class C2 {  
4046   string p2;  
4047 };
```

```
4048     class C3 { };
4049     [Association] class A12 {
4050         C1 REF x;
4051         C2 REF r; // same name as A13::r
4052     };
4053     [Association] class A13 {
4054         C1 REF y;
4055         C3 REF r; // same name as A12::r
4056     };
```

4057
4058
4059
4060**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

4061

4062 Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Document Set*,
4063 Rational Software Corporation, 1996, <http://www.rational.com/uml>.

4064 James O. Coplein, Douglas C. Schmidt (eds). *Pattern Languages of Program Design*, Addison-Wesley,
4065 Reading Mass., 1995.

4066 Georges Gardarin and Patrick Valduriez, *Relational Databases and Knowledge Bases*, Addison Wesley,
4067 1989.

4068 Gerald M. Weinberg (1975) *An Introduction to General Systems Thinking* (1975 ed., Wiley-Interscience)
4069 (2001 ed. Dorset House).

4070 Unicode Consortium, *The Unicode Standard*, Version 2.0, Addison-Wesley, 1996.

4071