



1
2
3
4
5

Document Number: DSP0004

Date: 2010-03-17

Version: 2.6.0

6 **Common Information Model (CIM) Infrastructure**

7 **Document Type: Specification**
8 **Document Status: DMTF Standard**
9 **Document Language: E**

10 Copyright Notice

11 Copyright © 1997, 2010 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26 withdrawn or modified after publication, and shall be indemnified and held harmless by any party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 **Trademarks**

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

35

36

CONTENTS

38	Foreword	6
39	Introduction	7
40	Document Conventions	7
41	CIM Management Schema	8
42	Core Model	8
43	Common Model	9
44	Extension Schema	9
45	CIM Implementations	9
46	CIM Implementation Conformance	10
47	1 Scope	11
48	2 Normative References	11
49	3 Terms and Definitions	13
50	4 Symbols and Abbreviated Terms	25
51	5 Meta Schema	26
52	5.1 Definition of the Meta Schema	26
53	5.1.1 Formal Syntax used in Descriptions	29
54	5.1.2 CIM Meta-Elements	30
55	5.2 Data Types	46
56	5.2.1 UCS and Unicode	47
57	5.2.2 String Type	48
58	5.2.3 Char16 Type	49
59	5.2.4 Datetime Type	49
60	5.2.5 Indicating Additional Type Semantics with Qualifiers	54
61	5.2.6 Comparison of Values	54
62	5.3 Supported Schema Modifications	55
63	5.3.1 Schema Versions	63
64	5.4 Class Names	65
65	5.5 Qualifiers	65
66	5.5.1 Qualifier Concept	65
67	5.5.2 Meta Qualifiers	69
68	5.5.3 Standard Qualifiers	69
69	5.5.4 Optional Qualifiers	90
70	5.5.5 User-defined Qualifiers	94
71	5.5.6 Mapping Entities of Other Information Models to CIM	94
72	6 Managed Object Format	98
73	6.1 MOF Usage	98
74	6.2 Class Declarations	99
75	6.3 Instance Declarations	99
76	7 MOF Components	99
77	7.1 Keywords	99
78	7.2 Comments	99
79	7.3 Validation Context	99
80	7.4 Naming of Schema Elements	99
81	7.5 Class Declarations	100
82	7.5.1 Declaring a Class	100
83	7.5.2 Subclasses	101
84	7.5.3 Default Property Values	101
85	7.5.4 Key Properties	102
86	7.5.5 Static Properties	103
87	7.6 Association Declarations	103
88	7.6.1 Declaring an Association	103

89	7.6.2	Subassociations.....	103
90	7.6.3	Key References and Properties in Associations.....	104
91	7.6.4	Weak Associations and Propagated Keys.....	104
92	7.6.5	Object References.....	107
93	7.7	Qualifiers.....	108
94	7.7.1	Qualifier Type.....	108
95	7.7.2	Qualifier Value.....	108
96	7.8	Instance Declarations.....	111
97	7.8.1	Instance Aliasing.....	113
98	7.8.2	Arrays.....	114
99	7.9	Method Declarations.....	116
100	7.9.1	Static Methods.....	117
101	7.10	Compiler Directives.....	117
102	7.11	Value Constants.....	117
103	7.11.1	String Constants.....	117
104	7.11.2	Character Constants.....	118
105	7.11.3	Integer Constants.....	118
106	7.11.4	Floating-Point Constants.....	119
107	7.11.5	Object Reference Constants.....	119
108	7.11.6	NULL.....	119
109	8	Naming.....	119
110	8.1	CIM Namespaces.....	120
111	8.2	Naming CIM Objects.....	120
112	8.2.1	Object Paths.....	120
113	8.2.2	Object Path for Namespace Objects.....	121
114	8.2.3	Object Path for Qualifier Type Objects.....	122
115	8.2.4	Object Path for Class Objects.....	123
116	8.2.5	Object Path for Class Objects.....	123
117	8.2.6	Matching CIM Names.....	124
118	8.3	Identity of CIM Objects.....	125
119	8.4	Requirements on Specifications Using Object Paths.....	125
120	8.5	Object Paths Used in CIM MOF.....	125
121	8.6	Mapping CIM Naming and Native Naming.....	126
122	8.6.1	Native Name Contained in Opaque CIM Key.....	127
123	8.6.2	Native Storage of CIM Name.....	127
124	8.6.3	Translation Table.....	127
125	8.6.4	No Mapping.....	127
126	9	Mapping Existing Models into CIM.....	127
127	9.1	Technique Mapping.....	127
128	9.2	Recast Mapping.....	128
129	9.3	Domain Mapping.....	131
130	9.4	Mapping Scratch Pads.....	131
131	10	Repository Perspective.....	131
132	10.1	DMTF MIF Mapping Strategies.....	133
133	10.2	Recording Mapping Decisions.....	133
134	ANNEX A (normative)	MOF Syntax Grammar Description.....	136
135	ANNEX B (informative)	CIM Meta Schema.....	142
136	ANNEX C (normative)	Units.....	163
137	C.1	Programmatic Units.....	163
138	C.2	Value for Units Qualifier.....	167
139	ANNEX D (informative)	UML Notation.....	170
140	ANNEX E (informative)	Guidelines.....	172
141	E.1	SQL Reserved Words.....	172
142	ANNEX F (normative)	EmbeddedObject and EmbeddedInstance Qualifiers.....	175

143	F.1	Encoding for MOF	175
144	F.2	Encoding for CIM Protocols	176
145		ANNEX G (informative) Schema Errata	177
146		ANNEX H (informative) Ambiguous Property and Method Names	179
147		ANNEX I (informative) OCL Considerations	182
148		ANNEX J (informative) Change Log	184
149		Bibliography	186
150			
151		Figures	
152		Figure 1 – Four Ways to Use CIM	9
153		Figure 2 – CIM Meta Schema	28
154		Figure 3 – Example with Two Weak Associations and Propagated Keys	105
155		Figure 4 – General Component Structure of Object Path	121
156		Figure 5 – Component Structure of Object Path for Namespaces	122
157		Figure 6 – Component Structure of Object Path for Qualifier Types	122
158		Figure 7 – Component Structure of Object Path for Classes	123
159		Figure 8 – Component Structure of Object Path for Instances	124
160		Figure 9 – Technique Mapping Example	128
161		Figure 10 – MIF Technique Mapping Example	128
162		Figure 11 – Recast Mapping	129
163		Figure 12 – Repository Partitions	132
164		Figure 13 – Homogeneous and Heterogeneous Export	134
165		Figure 14 – Scratch Pads and Mapping	134
166			
167		Tables	
168		Table 1 – Standards Bodies	11
169		Table 2 – Intrinsic Data Types	47
170		Table 3 – Compatibility of Schema Modifications	57
171		Table 4 – Compatibility of Qualifier Type Modifications	62
172		Table 5 – Changes that Increment the CIM Schema Major Version Number	64
173		Table 6 – Defined Qualifier Scopes	66
174		Table 7 – Defined Qualifier Flavors	67
175		Table 8 – Example for Mapping a String Format Based on the General Mapping String Format	96
176		Table 9 – UML Cardinality Notations	108
177		Table 10 – Standard Compiler Directives	117
178		Table 11 – Domain Mapping Example	131
179		Table C-1 – Base Units for Programmatic Units	165
180		Table D-1 – Diagramming Notation and Interpretation Summary	170
181			

182

Foreword

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

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

187 **Acknowledgments**

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

189 Editor:

- 190 • Lawrence Lamers – VMware

191 Contributors:

- 192 • Jeff Piazza – HP
- 193 • Andreas Maier – IBM
- 194 • George Ericson – EMC
- 195 • Jim Davis – WBEM Solutions
- 196 • Karl Schopmeyer – Inova Development

197

Introduction

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

206 Document Conventions

207 Typographical Conventions

208 The following typographical conventions are used in this document:

- 209 • Document titles are marked in *italics*.
- 210 • Important terms that are used for the first time are marked in *italics*.
- 211 • ABNF rules, OCL text and CIM MOF text are in `monospaced font`.

212 ABNF Usage Conventions

213 Format definitions in this document are specified using ABNF (see [RFC5234](#)), with the following
214 deviations:

- 215 • Literal strings are to be interpreted as case-sensitive UCS/Unicode characters, as opposed to
216 the definition in [RFC5234](#) that interprets literal strings as case-insensitive US-ASCII characters.
- 217 • By default, ABNF rules (including literals) are to be assembled without inserting any additional
218 whitespace characters, consistent with [RFC5234](#). If an ABNF rule states "whitespace allowed",
219 zero or more of the following whitespace characters are allowed between any ABNF rules
220 (including literals) that are to be assembled:
 - 221 – U+0009 (horizontal tab)
 - 222 – U+000A (linefeed, newline)
 - 223 – U+000C (form feed)
 - 224 – U+000D (carriage return)
 - 225 – U+0020 (space)
- 226 • In previous versions of this document, the vertical bar (|) was used to indicate a choice. Starting
227 with version 2.6 of this document, the forward slash (/) is used to indicate a choice, as defined in
228 [RFC5234](#).

229 Deprecated Material

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

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

237 The following typographical convention indicates deprecated material:

238 **DEPRECATED**

239 Deprecated material appears here.

240 **DEPRECATED**

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

243 **Experimental Material**

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

249 The following typographical convention indicates experimental material:

250 **EXPERIMENTAL**

251 Experimental material appears here.

252 **EXPERIMENTAL**

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

255 **CIM Management Schema**

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

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

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

264 **Core Model**

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

269 **Common Model**

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

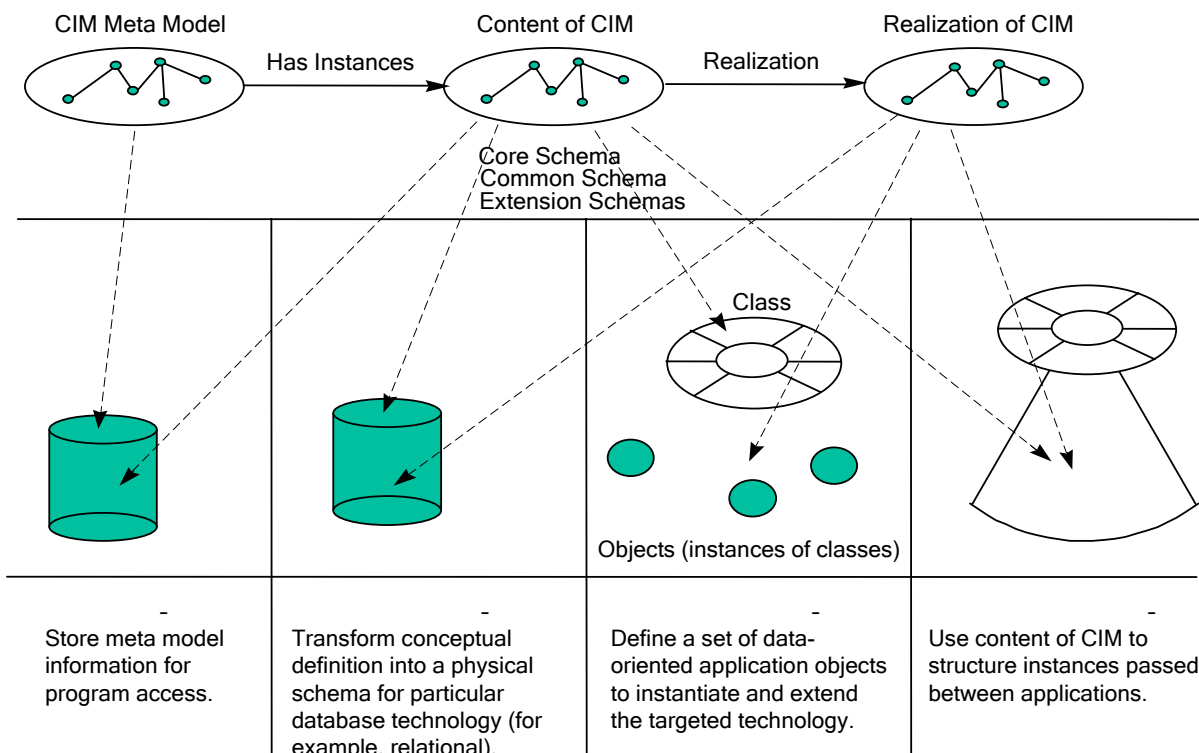
276 The common model is an information model common to particular management areas but independent of
 277 a particular technology or implementation. The common areas are systems, applications, networks, and
 278 devices. The information model is specific enough to provide a basis for developing management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

282 **Extension Schema**

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

286 **CIM Implementations**

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



290

291

Figure 1 – Four Ways to Use CIM

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

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

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

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

308 **CIM Implementation Conformance**

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

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

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

317 Common Information Model (CIM) Infrastructure

318 1 Scope

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

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

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

329 2 Normative References

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

334 Table 1 shows standards bodies and their web sites.

335 **Table 1 – Standards Bodies**

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

336

- 337 ANSI/IEEE 754-1985, *IEEE® Standard for BinaryFloating-Point Arithmetic*, August 1985
338 http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=30711
- 339 DMTF DSP0207, *WBEM URI Mapping Specification*, Version 1.0
340 http://www.dmtf.org/standards/published_documents/DSP0207_1.0.pdf
- 341 DMTF DSP4004, *DMTF Release Process*, Version 2.2
342 http://www.dmtf.org/standards/published_documents/DSP4004_2.2.pdf
- 343 EIA-310, *Cabinets, Racks, Panels, and Associated Equipment*
344 <http://electronics.ihs.com/collections/abstracts/eia-310.htm>
- 345 IEEE Std 1003.1, 2004 Edition, *Standard for information technology - portable operating system interface*
346 *(POSIX). Shell and utilities*
347 http://www.unix.org/version3/ieee_std.html
- 348 IETF RFC3986, *Uniform Resource Identifiers (URI): Generic Syntax*, August 1998
349 <http://tools.ietf.org/html/rfc2396>
- 350 IETF RFC5234, *Augmented BNF for Syntax Specifications: ABNF*, January 2008
351 <http://tools.ietf.org/html/rfc5234>
- 352 ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*
353 <http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype>
- 354 ISO 639-1:2002, *Codes for the representation of names of languages — Part 1: Alpha-2 code*
355 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109
- 356 ISO 639-2:1998, *Codes for the representation of names of languages — Part 2: Alpha-3 code*
357 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=4767
- 358 ISO 639-3:2007, *Codes for the representation of names of languages — Part 3: Alpha-3 code for*
359 *comprehensive coverage of languages*
360 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39534
- 361 ISO 1000:1992, *SI units and recommendations for the use of their multiples and of certain other units*
362 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=5448
- 363 ISO 3166-1:2006, *Codes for the representation of names of countries and their subdivisions — Part 1:*
364 *Country codes*
365 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39719
- 366 ISO 3166-2:2007, *Codes for the representation of names of countries and their subdivisions — Part 2:*
367 *Country subdivision code*
368 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39718
- 369 ISO 3166-3:1999, *Codes for the representation of names of countries and their subdivisions — Part 3:*
370 *Code for formerly used names of countries*
371 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=2130
- 372 ISO 8601:2004 (E), *Data elements and interchange formats – Information interchange — Representation*
373 *of dates and times*
374 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40874
- 375 ISO/IEC 9075-10:2003, *Information technology — Database languages — SQL — Part 10: Object*
376 *Language Bindings (SQL/OLB)*
377 http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=34137

- 378 ISO/IEC 10165-4:1992, *Information technology — Open Systems Interconnection – Structure of*
379 *management information — Part 4: Guidelines for the definition of managed objects (GDMO)*
380 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=18174
- 381 ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*
382 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c039921_ISO_IEC_10646_2003(E).zip)
- 383 ISO/IEC 10646:2003/Amd 1:2005, *Information technology — Universal Multiple-Octet Coded Character*
384 *Set (UCS) — Amendment 1: Glagolitic, Coptic, Georgian and other characters*
385 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005\(E\).](http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip)
386 [zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c040755_ISO_IEC_10646_2003_Amd_1_2005(E).zip)
- 387 ISO/IEC 10646:2003/Amd 2:2006, *Information technology — Universal Multiple-Octet Coded Character*
388 *Set (UCS) — Amendment 2: N'Ko, Phags-pa, Phoenician and other characters*
389 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006\(E\).](http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip)
390 [zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip)
- 391 ISO/IEC 14651:2007, *Information technology — International string ordering and comparison — Method*
392 *for comparing character strings and description of the common template tailorable ordering*
393 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c044872_ISO_IEC_14651_2007\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c044872_ISO_IEC_14651_2007(E).zip)
- 394 ISO/IEC 14750:1999, *Information technology — Open Distributed Processing — Interface Definition*
395 *Language*
396 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25486
- 397 ITU X.501, *Information Technology — Open Systems Interconnection — The Directory: Models*
398 <http://www.itu.int/rec/T-REC-X.501/en>
- 399 ITU X.680 (07/02), *Information technology — Abstract Syntax Notation One (ASN.1): Specification of*
400 *basic notation*
401 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf>
- 402 OMG, *Object Constraint Language, Version 2.0*
403 <http://www.omg.org/cgi-bin/doc?formal/2006-05-01>
- 404 OMG, *Unified Modeling Language: Superstructure, Version 2.1.1*
405 <http://www.omg.org/cgi-bin/doc?formal/07-02-05>
- 406 The Unicode Consortium, *The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization*
407 *Forms*
408 <http://www.unicode.org/reports/tr15/>
- 409 W3C, *Namespaces in XML*, W3C Recommendation, 14 January 1999
410 <http://www.w3.org/TR/REC-xml-names>

411 **3 Terms and Definitions**

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

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

420 The terms "clause," "subclause," "paragraph," and "annex" in this document are to be interpreted as
421 described in [ISO/IEC Directives, Part 2](#), Clause 5.

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

425 The following additional terms are used in this document.

426 **3.1**

427 **address**

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

430 More specific kinds of addresses are object paths.

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

433 **3.2**

434 **aggregation**

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

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

441 **3.3**

442 **ancestor**

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

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

448 **3.4**

449 **ancestry**

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

452 A schema element is not considered part of its ancestry.

453 **3.5**

454 **arity**

455 the number of references exposed by an association class

456 **3.6**

457 **association, CIM association**

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

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

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

463 A CIM association is a UML association class. Each has the aspects of both a UML association and a
464 UML class, which may expose ordinary properties and methods and may be part of a class inheritance
465 hierarchy. The references belonging to a CIM association belong to it and are also exposed as part of the

466 association and not as parts of the associated classes. The term "association class" is sometimes used
467 instead of the term "association" when the class aspects of the element are being emphasized.

468 Aggregations and compositions are special kinds of associations.

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

474 In a schema, associations are special kinds of schema elements.

475 In the CIM meta-model, associations are represented by the meta-element named "Association".

476 3.7

477 **association end**

478 a synonym for the reference defined in an association

479 3.8

480 **cardinality**

481 the number of instances in a set

482 **DEPRECATED**

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

485 **DEPRECATED**

486 3.9

487 **Common Information Model**

488 **CIM**

489 CIM (Common Information Model) is:

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

492 3.10

493 **CIM schema**

494 the schema published by the DMTF that defines the Common Information Model

495 It is divided into a core model and a common model. Extension schemas are defined outside of the DMTF
496 and are not considered part of the CIM schema.

497 3.11

498 **CIM client**

499 a role responsible for originating CIM operations for processing by a CIM server

500 This definition does not imply any particular implementation architecture or scope, such as a client library
501 component or an entire management application.

502 3.12

503 **CIM listener**

504 a role responsible for processing CIM indications originated by a CIM server

505 This definition does not imply any particular implementation architecture or scope, such as a standalone
506 demon component or an entire management application.

- 507 **3.13**
508 **CIM operation**
509 an interaction within a CIM protocol that is originated by a CIM client and processed by a CIM server
- 510 **3.14**
511 **CIM protocol**
512 a protocol that is used between CIM client, CIM server and CIM listener
513 This definition does not imply any particular communication protocol stack, or even that the protocol
514 performs a remote communication.
- 515 **3.15**
516 **CIM server**
517 a role responsible for processing CIM operations originated by a CIM client and for originating CIM
518 indications for processing by a CIM listener
519 This definition does not imply any particular implementation architecture, such as a separation into a
520 CIMOM and provider components.
- 521 **3.16**
522 **class, CIM class**
523 a common type for a set of instances that support the same features
524 A class is defined in a schema and models an aspect of a managed object. For a full definition, see
525 5.1.2.7.
526 For example, a class named "ACME_Modem" may represent a common type for instances of modems
527 and may define common features such as a property named "ActualSpeed" to represent the actual
528 modem speed.
529 Special kinds of classes are ordinary classes, association classes and indication classes.
530 In a CIM server, classes are special kinds of objects. The term "class object" (i.e., object of class type) is
531 sometimes used to emphasize that. The address of such class objects is termed "class path".
532 In a schema, classes are special kinds of schema elements.
533 In the CIM meta-model, classes are represented by the meta-element named "Class".
- 534 **3.17**
535 **class declaration**
536 the definition (or specification) of a class
537 For example, a class that is accessible through a CIM server can be retrieved by a CIM client. What the
538 CIM client receives as a result is actually the class declaration. Although unlikely, the class accessible
539 through the CIM server may already have changed its definition by the time the CIM client receives the
540 class declaration. Similarly, when a class accessible through a CIM server is being modified through a
541 CIM operation, one input parameter might be a class declaration that is used during the processing of the
542 CIM operation to change the class.
- 543 **3.18**
544 **class path**
545 a special kind of object path addressing a class that is accessible through a CIM server
- 546 **3.19**
547 **class origin**
548 the class origin of a feature is the class defining the feature
- 549 **3.20**
550 **common model**
551 the subset of the CIM Schema that is specific to particular domains
552 It is derived from the core model and is actually a collection of models, including (but not limited to) the
553 System model, the Application model, the Network model, and the Device model.

554 **3.21**555 **composition**

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

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

562 **3.22**563 **core model**

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

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

567 **3.23**568 **creation class**

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

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

573 **3.24**574 **domain**

575 an area of management or expertise

576 DEPRECATED

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

582 DEPRECATED583 **3.25**584 **effective qualifier value**

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

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

595 **3.26**596 **element**

597 a synonym for schema element

- 598 **3.27**
599 **embedded class**
600 a class declaration that is embedded in the value of a property, parameter or method return value
- 601 **3.28**
602 **embedded instance**
603 an instance declaration that is embedded in the value of a property, parameter or method return value
- 604 **3.29**
605 **embedded object**
606 an embedded class or embedded instance
- 607 **3.30**
608 **explicit qualifier**
609 a qualifier type declared separately from its usage on schema elements
610 See also implicit qualifier.
- 611 **3.31**
612 **extension schema**
613 a schema not owned by the DMTF whose classes are derived from the classes in the CIM Schema
- 614 **3.32**
615 **feature**
616 a property or method defined in a class
617 A feature is exposed if it is available to consumers of a class. The set of features exposed by a class is
618 the union of all features defined in the class and its ancestry. In the case where a feature overrides a
619 feature, the combined effects are exposed as a single feature.
- 620 **3.33**
621 **flavor**
622 meta-data on a qualifier type that defines the rules for propagation, overriding and translatability of
623 qualifiers
624 For example, the Key qualifier has the flavors ToSubclass and DisableOverride, meaning that the qualifier
625 value gets propagated to subclasses and these subclasses cannot override it.
- 626 **3.34**
627 **implicit qualifier**
628 a qualifier type declared as part of the declaration of a schema element
629 See also explicit qualifier.
-
- 630 **DEPRECATED**
- 631 The concept of implicitly defined qualifier types (i.e., implicit qualifiers) is deprecated. See 5.1.2.16 for
632 details.
- 633 **DEPRECATED**
-
- 634 **3.35**
635 **indication, CIM indication**
636 a special kind of class that expresses the notification about an event that occurred
637 Indications are raised based on a trigger that defines the condition under which an event causes an
638 indication to be raised. Events may be related to objects accessible in a CIM server, such as the creation,

639 modification, deletion of or access to an object, or execution of a method on the object. Events may also
640 be related to managed objects, such as alerts or errors.

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

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

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

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

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

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

650 3.36

651 inheritance

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

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

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

658 3.37

659 instance, CIM instance

660 This term has two (different) meanings:

661 1) As instance of a class:

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

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

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

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

669 2) As instance of a meta-element:

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

673 3.38

674 instance path

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

676 3.39

677 instance declaration

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

680 For example, an instance that is accessible through a CIM server can be retrieved by a CIM client. What
681 the CIM client receives as a result, is actually an instance declaration. The instance itself may already
682 have changed its property values by the time the CIM client receives the instance declaration. Similarly,
683 when an instance that is accessible through a CIM server is being modified through a CIM operation, one

684 input parameter might be an instance declaration that specifies the intended new property values for the
685 instance.

686 **3.40**

687 **key**

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

691 Also, shorthand for the term "key property".

692 **3.41**

693 **managed object**

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

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

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

698 **3.42**

699 **meta-element**

700 an entity in a meta-model

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

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

705 **3.43**

706 **meta-model**

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

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

711 **3.44**

712 **meta-relationship**

713 a relationship between two entities in a meta-model

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

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

717 **3.45**

718 **meta-schema**

719 a synonym for meta-model

720 **3.46**

721 **method, CIM method**

722 a behavioral feature of a class

723 Methods can be invoked to produce the associated behavior.

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

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

- 727 **3.47**
728 **model**
729 a set of classes that model a specific domain
730 A schema may contain multiple models (that is the case in the CIM Schema), but a particular domain
731 could also be modeled using multiple schemas, in which case a model would consist of multiple schemas.
- 732 **3.48**
733 **model path**
734 the part of an object path that identifies the object within the namespace
- 735 **3.49**
736 **multiplicity**
737 The multiplicity of an association end is the allowable range for the number of instances that may be
738 associated to each instance referenced by each of the other ends of the association. The multiplicity is
739 defined on a reference using the Min and Max qualifiers.
- 740 **3.50**
741 **namespace, CIM namespace**
742 a special kind of object that is accessible through a CIM server that represents a naming space for
743 classes, instances and qualifier types
- 744 **3.51**
745 **namespace path**
746 a special kind of object path addressing a namespace that is accessible through a CIM server
747 Also, the part of an instance path, class path and qualifier type path that addresses the namespace.
- 748 **3.52**
749 **name**
750 an identifier that each element or meta-element has in order to identify it in some scope
-
- 751 **DEPRECATED**
752 The use of the term "name" for the address of an object that is accessible through a CIM server is
753 deprecated. The term "object path" should be used instead.
- 754 **DEPRECATED**
-
- 755 **3.53**
756 **object, CIM object**
757 a class, instance, qualifier type or namespace that is accessible through a CIM server
758 An object may be addressable, i.e., have an object path. Embedded objects are objects that are not
759 addressable; they are accessible indirectly through their embedding property, parameter or method return
760 value. Instances of indications are objects that are not addressable either, as they are not accessible
761 through a CIM server at all and only exist in the protocol message in which they are being delivered.
-
- 762 **DEPRECATED**
763 The term "object" has historically be used to mean just "class or instance". This use of the term "object" is
764 deprecated. If a restriction of the term "object" to mean just "class or instance" is intended, this is now
765 stated explicitly.
- 766 **DEPRECATED**
-

- 767 **3.54**
768 **object path**
769 the address of an object that is accessible through a CIM server
770 An object path consists of a namespace path (addressing the namespace) and optionally a model path
771 (identifying the object within the namespace).
- 772 **3.55**
773 **ordinary class**
774 a class that is neither an association class nor an indication class
- 775 **3.56**
776 **ordinary property**
777 a property that is not a reference
- 778 **3.57**
779 **override**
780 a relationship between like-named elements of the same type of meta-element in an inheritance
781 hierarchy, where the overriding element in a subclass redefines the overridden element in a superclass
782 The purpose of an override relationship is to refine the definition of an element in a subclass.
783 For example, a class ACME_Device may define a string typed property Status that may have the values
784 "powersave", "on", or "off". A class ACME_Modem, subclass of ACME_Device, may override the Status
785 property to have only the values "on" or "off", but not "powersave".
- 786 **3.58**
787 **parameter, CIM parameter**
788 a named and typed argument passed in and out of methods
789 The return value of a method is not considered a parameter; instead it is considered part of the method.
790 In a schema, parameters are special kinds of schema elements.
791 In the CIM meta-model, parameters are represented by the meta-element named "Parameter".
- 792 **3.59**
793 **polymorphism**
794 the ability of an instance to be of a class and all of its subclasses
795 For example, a CIM operation may enumerate all instances of class ACME_Device. If the instances
796 returned may include instances of subclasses of ACME_Device, then that CIM operation is said to
797 implement polymorphic behavior.
- 798 **3.60**
799 **propagation**
800 the ability to derive a value of one property from the value of another property
801 CIM supports propagation via either PropertyConstraint qualifiers utilizing a derivation constraint or via
802 weak associations.
- 803 **3.61**
804 **property, CIM property**
805 a named and typed structural feature of a class
806 Name, data type, default value and other information about the property are defined in a class. Properties
807 have values that are available in the instances of a class. The values of its properties may be used to
808 characterize an instance.
809 For example, a class ACME_Device may define a string typed property named "Status". In an instance of
810 class ACME_Device, the Status property may have a value "on".
811 Special kinds of properties are ordinary properties and references.
812 In a schema, properties are special kinds of schema elements.

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

814 **3.62**

815 **qualified element**

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

817 For example, the term "qualified element" in the description of the Counter qualifier refers to any property

818 (or other kind of schema element) that has the Counter qualifier specified on it.

819 **3.63**

820 **qualifier, CIM qualifier**

821 a named value used to characterize schema elements

822 Qualifier values may change the behavior or semantics of the qualified schema element. Qualifiers can

823 be regarded as metadata that is attached to the schema elements. The scope of a qualifier determines on

824 which kinds of schema elements a specific qualifier can be specified.

825 For example, if property ACME_Modem.Speed has the Key qualifier specified with a value of TRUE, this

826 characterizes the property as a key property for the class.

827 **3.64**

828 **qualifier type**

829 a common type for a set of qualifiers

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

831 termed "qualifier type path".

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

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

834 **3.65**

835 **qualifier type declaration**

836 the definition (or specification) of a qualifier type

837 For example, a qualifier type object that is accessible through a CIM server can be retrieved by a CIM

838 client. What the CIM client receives as a result, is actually a qualifier type declaration. Although unlikely,

839 the qualifier type itself may already have changed its definition by the time the CIM client receives the

840 qualifier type declaration. Similarly, when a qualifier type that is accessible through a CIM server is being

841 modified through a CIM operation, one input parameter might be a qualifier type declaration that is used

842 during the processing of the operation to change the qualifier type.

843 **3.66**

844 **qualifier type path**

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

846 **3.67**

847 **qualifier value**

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

849 value, or the default value

850 **3.68**

851 **reference, CIM reference**

852 an association end

853 References are special kinds of properties that reference an instance.

854 The value of a reference is an instance path. The type of a reference is a class of the referenced

855 instance. The referenced instance may be of a subclass of the class specified as the type of the

856 reference.

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

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

- 859 **3.69**
860 **schema**
861 a set of classes with a single defining authority or owning organization
862 In the CIM meta-model, schemas are represented by the meta-element named "Schema".
- 863 **3.70**
864 **schema element**
865 a specific class, property, method or parameter
866 For example, a class ACME_C1 or a property P1 are schema elements.
- 867 **3.71**
868 **scope**
869 part of a qualifier type, indicating the meta-elements on which the qualifier can be specified
870 For example, the Abstract qualifier has scope class, association and indication, meaning that it can be
871 specified only on ordinary classes, association classes, and indication classes.
- 872 **3.72**
873 **scoping object, scoping instance, scoping class**
874 a scoping object provides context for a set of other objects
875 A specific example is an object (class or instance) that propagates some or all of its key properties to a
876 weak object, along a weak association.
- 877 **3.73**
878 **signature**
879 a method name together with the type of its return value and the set of names and types of its parameters
- 880 **3.74**
881 **subclass**
882 See inheritance.
- 883 **3.75**
884 **superclass**
885 See inheritance.
- 886 **3.76**
887 **top-level object**
-
- 888 **DEPRECATED**
889 The use of the terms "top-level object" or "TLO" for an object that has no scoping object is deprecated.
890 Use phrases like "an object that has no scoping object", instead.
- 891 **DEPRECATED**
-
- 892 **3.77**
893 **trigger**
894 a condition that when true, expresses the occurrence of an event
- 895 **3.78**
896 **weak object, weak instance, weak class**
897 an object (class or instance) that gets some or all of its key properties propagated from a scoping object,
898 along a weak association

899 **3.79**

900 **weak association**

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

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

906 **4 Symbols and Abbreviated Terms**

907 The following abbreviations are used in this document.

908 **4.1**

909 **API**

910 application programming interface

911 **4.2**

912 **CIM**

913 Common Information Model

914 **4.3**

915 **DBMS**

916 Database Management System

917 **4.4**

918 **DMI**

919 Desktop Management Interface

920 **4.5**

921 **GDMO**

922 Guidelines for the Definition of Managed Objects

923 **4.6**

924 **HTTP**

925 Hypertext Transfer Protocol

926 **4.7**

927 **MIB**

928 Management Information Base

929 **4.8**

930 **MIF**

931 Management Information Format

932 **4.9**

933 **MOF**

934 Managed Object Format

935 **4.10**
936 **OID**
937 object identifier

938 **4.11**
939 **SMI**
940 Structure of Management Information

941 **4.12**
942 **SNMP**
943 Simple Network Management Protocol

944 **4.13**
945 **UML**
946 Unified Modeling Language

947 **5 Meta Schema**

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

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

955 **5.1 Definition of the Meta Schema**

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

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

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

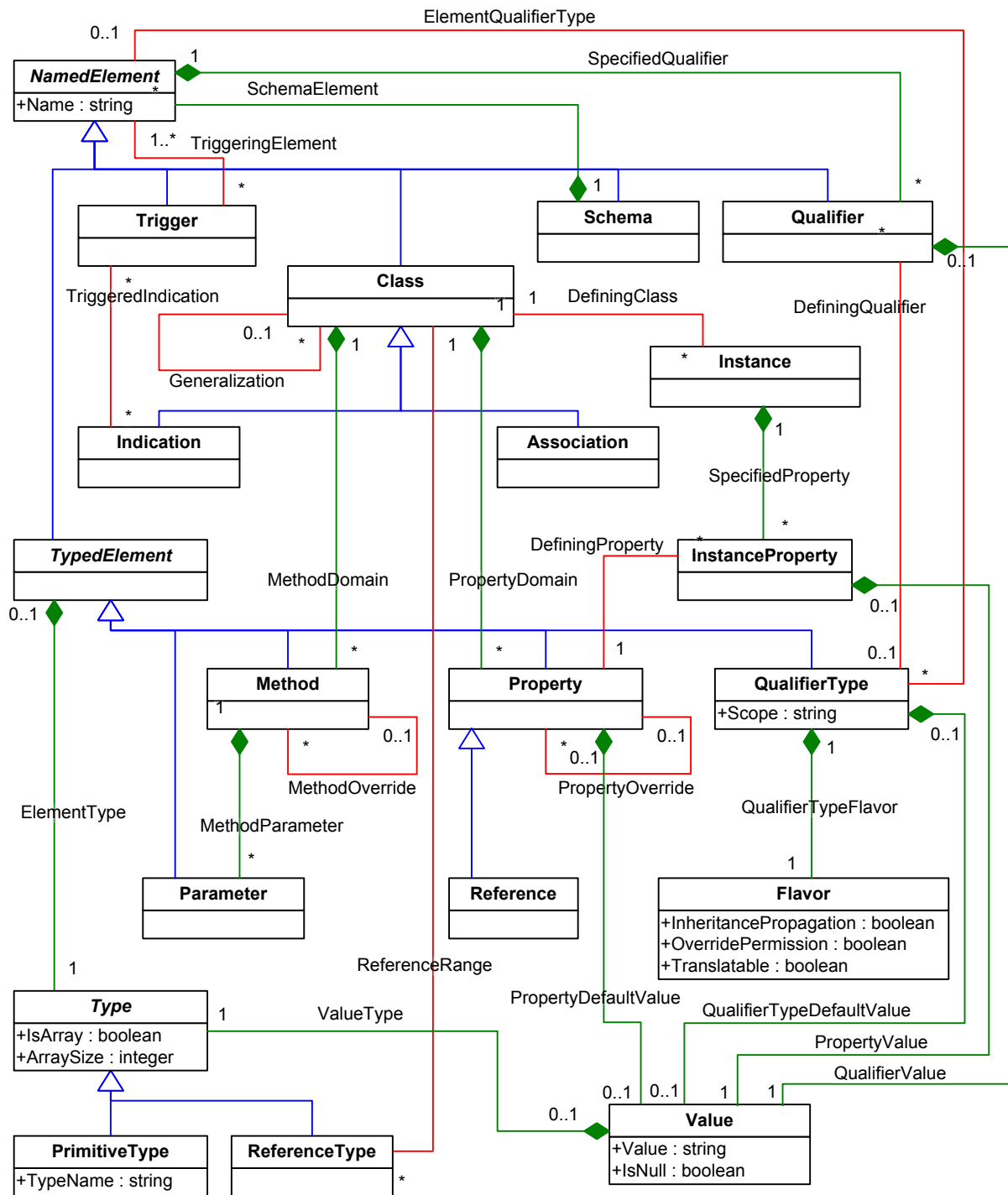
972 Languages defining CIM schemas and models (e.g., CIM Managed Object Format) shall use the meta-
973 schema defined in this subclause, or an equivalent meta-schema, as a basis.

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

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

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

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



989

990

Figure 2 – CIM Meta Schema

991

992

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

993 5.1.1 Formal Syntax used in Descriptions

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

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

```

998 attribute-format = attr-name ":" attr-type ( "[" attr-multiplicity "]" )
999     ; the format used to describe the attributes of CIM meta-elements
1000
1001 attr-name = IDENTIFIER
1002     ; the name of the attribute
1003
1004 attr-type = type
1005     ; the datatype of the attribute
1006
1007 type = "string"    ; a string of UCS characters of arbitrary length
1008     / "boolean"   ; a boolean value
1009     / "integer"   ; a signed 64-bit integer value
1010
1011 attr-multiplicity = cardinality-format
1012     ; the multiplicity of the attribute. The default multiplicity is 1
  
```

1013 Descriptions of association ends use the `association-end-format` ABNF rule:

```

1014 association-end-format = other-role ":" other-element "[" other-cardinality "]"
1015     ; the format used to describe association ends of associations
1016     ; between CIM meta-elements
1017
1018 other-role = IDENTIFIER
1019     ; the role of the association end (on this side of the relationship)
1020     ; that is referencing the associated meta-element
1021
1022 other-element = IDENTIFIER
1023     ; the name of the associated meta-element
1024
1025 other-cardinality = cardinality-format
1026     ; the cardinality of the associated meta-element
1027
1028 cardinality-format = positiveIntegerValue           ; exactly that
1029     / "*"                                             ; zero to any
1030     / integerValue ".." positiveIntegerValue       ; min to max
1031     / integerValue ".." "*"                         ; min to any
1032     ; format of a cardinality specification
1033
1034 integerValue = decimalDigit *decimalDigit         ; no whitespace allowed
1035
1036 positiveIntegerValue = positiveDecimalDigit *decimalDigit ; no whitespace allowed
  
```

1037 5.1.2 CIM Meta-Elements

1038 5.1.2.1 NamedElement

1039 Abstract class for CIM elements, providing the ability for an element to have a name.

1040 Some kinds of elements provide the ability to have qualifiers specified on them, as described in
1041 subclasses of *NamedElement*.

1042 Generalization: None

1043 Non-default UML characteristics: isAbstract = true

1044 Attributes:

- 1045 • *Name* : string

1046 The name of the element. The format of the name is determined by subclasses of
1047 *NamedElement*.

1048 The names of elements shall be compared case-insensitively.

1049 Association ends:

- 1050 • *OwnedQualifier* : Qualifier [*] (composition *SpecifiedQualifier*, aggregating on its
1051 *OwningElement* end)

1052 The qualifiers specified on the element.

- 1053 • *OwningSchema* : Schema [1] (composition *SchemaElement*, aggregating on its
1054 *OwningSchema* end)

1055 The schema owning the element.

- 1056 • *Trigger* : Trigger [*] (association *TriggeringElement*)

1057 The triggers specified on the element.

- 1058 • *QualifierType* : QualifierType [*] (association *ElementQualifierType*)

1059 The qualifier types implicitly defined on the element.

1060 Note: Qualifier types defined explicitly are not associated to elements; they are global in the
1061 CIM namespace.

1062 DEPRECATED

1063 The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

1064 DEPRECATED

1065 Additional constraints:

- 1066 1) The value of *Name* shall not be NULL.

1067 5.1.2.2 TypedElement

1068 Abstract class for CIM elements that have a CIM data type.

1069 Not all kinds of CIM data types may be used for all kinds of typed elements. The details are determined
1070 by subclasses of *TypedElement*.

1071 Generalization: *NamedElement*

1072 Non-default UML characteristics: *isAbstract* = true

1073 Attributes: None

1074 Association ends:

- 1075 • *OwnedType* : Type [1] (composition *ElementType*, aggregating on its *OwningElement* end)

1076 The CIM data type of the element.

1077 Additional constraints: None

1078 5.1.2.3 Type

1079 Abstract class for any CIM data types, including arrays of such.

1080 Generalizations: None

1081 Non-default UML characteristics: *isAbstract* = true

1082 Attributes:

- 1083 • *isArray* : boolean

1084 Indicates whether the type is an array type. For details on arrays, see 7.8.2.

- 1085 • *ArraySize* : integer

1086 If the type is an array type, a non-NULL value indicates the size of a fixed-size array, and a
1087 NULL value indicates a variable-length array. For details on arrays, see 7.8.2.

1088 Association ends:

- 1089 • *OwningElement* : TypedElement [0..1] (composition *ElementType*, aggregating on its
1090 *OwningElement* end)

- 1091 • *OwningValue* : Value [0..1] (composition *ValueType*, aggregating on its *OwningValue* end)

1092 The element that has a CIM data type.

1093 Additional constraints:

1094 1) The value of *isArray* shall not be NULL.

1095 2) If the type is no array type, the value of *ArraySize* shall be NULL.

1096 Equivalent OCL class constraint:

```
1097 inv: self.isArray = false
1098     implies self.ArraySize.IsNull()
```

1099 3) A *Type* instance shall be owned by only one owner.

1100 Equivalent OCL class constraint:

```
1101 inv: self.ElementType[OwnedType].OwningElement->size() +
1102     self.ValueType[OwnedType].OwningValue->size() = 1
```

1103 5.1.2.4 PrimitiveType

1104 A CIM data type that is one of the intrinsic types defined in Table 2, excluding references.

1105 Generalization: *Type*

1106 Non-default UML characteristics: None

1107 Attributes:

- 1108 • *TypeName* : string

1109 The name of the CIM data type.

1110 Association ends: None

1111 Additional constraints:

- 1112 1) The value of *TypeName* shall follow the formal syntax defined by the `dataType` ABNF rule in
1113 ANNEX A.
- 1114 2) The value of *TypeName* shall not be NULL.
- 1115 3) This kind of type shall be used only for the following kinds of typed elements: *Method*,
1116 *Parameter*, ordinary *Property*, and *QualifierType*.

1117 Equivalent OCL class constraint:

```
1118 inv: let e : _NamedElement =
1119     self.ElementType[OwnedType].OwningElement
1120 in
1121     e.oclIsTypeOf(Method) or
1122     e.oclIsTypeOf(Parameter) or
1123     e.oclIsTypeOf(Property) or
1124     e.oclIsTypeOf(QualifierType)
```

1125 5.1.2.5 ReferenceType

1126 A CIM data type that is a reference, as defined in Table 2.

1127 Generalization: *Type*

1128 Non-default UML characteristics: None

1129 Attributes: None

1130 Association ends:

- 1131 • *ReferencedClass* : Class [1] (association *ReferenceRange*)

1132 The class referenced by the reference type.

1133 Additional constraints:

- 1134 1) This kind of type shall be used only for the following kinds of typed elements: *Parameter* and
1135 *Reference*.

1136 Equivalent OCL class constraint:

```
1137 inv: let e : NamedElement = /* the typed element */
1138     self.ElementType[OwnedType].OwningElement
1139 in
1140     e.oclIsTypeOf(Parameter) or
1141     e.oclIsTypeOf(Reference)
```

1142 2) When used for a *Reference*, the type shall not be an array.

1143 Equivalent OCL class constraint:

```
1144 inv: self.ElementType[OwnedType].OwningElement.
1145     oclIsTypeOf(Reference)
1146     implies
1147     self.IsArray = false
```

1148 5.1.2.6 Schema

1149 Models a CIM schema. A CIM schema is a set of CIM classes with a single defining authority or owning
1150 organization.

1151 Generalization: *NamedElement*

1152 Non-default UML characteristics: None

1153 Attributes: None

1154 Association ends:

- 1155 • *OwnedElement* : NamedElement [*] (composition *SchemaElement*, aggregating on its
1156 *OwningSchema* end)

1157 The elements owned by the schema.

1158 Additional constraints:

1159 1) The value of the *Name* attribute shall follow the formal syntax defined by the `schemaName`
1160 ABNF rule in ANNEX A.

1161 2) The elements owned by a schema shall be only of kind *Class*.

1162 Equivalent OCL class constraint:

```
1163 inv: self.SchemaElement[OwningSchema].OwnedElement.
1164     oclIsTypeOf(Class)
```

1165 5.1.2.7 Class

1166 Models a CIM class. A CIM class is a common type for a set of CIM instances that support the same
1167 features (i.e., properties and methods). A CIM class models an aspect of a managed element.

1168 Classes may be arranged in a generalization hierarchy that represents subtype relationships between
1169 classes. The generalization hierarchy is a rooted, directed graph and does not support multiple
1170 inheritance.

1171 A class may have methods, which represent their behavior, and properties, which represent the data
1172 structure of its instances.

1173 A class may participate in associations as the target of an association end owned by the association.

1174 A class may have instances.

1175 Generalization: *NamedElement*

1176 Non-default UML characteristics: None

1177 Attributes: None

1178 Association ends:

- 1179 • *OwnedProperty* : Property [*] (composition *PropertyDomain*, aggregating on its *OwningClass*
- 1180 end)

1181 The properties owned by the class.

- 1182 • *OwnedMethod* : Method [*] (composition *MethodDomain*, aggregating on its *OwningClass* end)

1183 The methods owned by the class.

- 1184 • *ReferencingType* : ReferenceType [*] (association *ReferenceRange*)

1185 The reference types referencing the class.

- 1186 • *SuperClass* : Class [0..1] (association *Generalization*)

1187 The superclass of the class.

- 1188 • *SubClass* : Class [*] (association *Generalization*)

1189 The subclasses of the class.

- 1190 • *Instance* : Instance [*] (association *DefiningClass*)

1191 The instances for which the class is their defining class.

1192 Additional constraints:

- 1193 1) The value of the *Name* attribute (i.e., the class name) shall follow the formal syntax defined by
- 1194 the `className` ABNF rule in ANNEX A.

1195 NOTE: The name of the schema containing the class is part of the class name.

- 1196 2) The class name shall be unique within the schema owning the class.

1197 5.1.2.8 Property

1198 Models a CIM property defined in a CIM class. A CIM property is the declaration of a structural feature of

1199 a CIM class, i.e., the data structure of its instances.

1200 Properties are inherited to subclasses such that instances of the subclasses have the inherited properties

1201 in addition to the properties defined in the subclass. The combined set of properties defined in a class

1202 and properties inherited from superclasses is called the properties exposed by the class.

1203 Classes that define a property without overriding an inherited property of the same name, expose two

1204 properties with that name. This is an undesirable situation since the resolution of property names to the

1205 actual properties is undefined in this document.

1206 DEPRECATED

1207 Within a single given schema (as defined in 5.1.2.6), the definition of properties without overriding

1208 inherited properties of the same name defined in a class of the same schema is deprecated. The

1209 deprecation only applies to the act of establishing that scenario, not necessarily to any schema elements

1210 that are involved.

1211 DEPRECATED

1212 Between an underlying schema (e.g., the DMTF published CIM schema) and a derived schema (e.g., a

1213 vendor schema), the definition of properties in the derived schema without overriding inherited properties

1214 of the same name defined in a class of the underlying schema may occur if both schemas are updated

1215 independently. Therefore, care should be exercised by the owner of the derived schema when moving to
1216 a new release of the underlying schema in order to avoid this situation.

1217 A class defining a property may indicate that the property overrides an inherited property. In this case, the
1218 class exposes only the overriding property. The characteristics of the overriding property are formed by
1219 using the characteristics of the overridden property as a basis, changing them as defined in the overriding
1220 property, within certain limits as defined in section "Additional constraints".

1221 If a property defines a default value, that default value represents an initialization constraint for the
1222 property. Initialization constraints for properties may also be specified via the *PropertyConstraint* qualifier
1223 (see 5.5.3.39). An initialization constraint determines the initial value of the property in new CIM
1224 instances. If no initialization constraint is defined for a property, its initial value in new CIM instances is
1225 undefined at the level of the schema, i.e., there is no implied initialization constraint of NULL.

1226 Other specifications may define additional means to determine the initial value of a property in new CIM
1227 instances; for example, management profiles may define initialization constraints, or operation
1228 specifications may define that operations that cause new CIM instances to come into existence support
1229 the ability to override the schema defined initialization constraints.

1230 Default values defined on properties in a class propagate to overriding properties in its subclasses. The
1231 value of the *PropertyConstraint* qualifier also propagates to overriding properties in subclasses, as
1232 defined in its qualifier type.

1233 Generalization: *TypedElement*

1234 Non-default UML characteristics: None

1235 Attributes: None.

1236 Association ends:

1237 • *OwningClass* : Class [1] (composition *PropertyDomain*, aggregating on its *OwningClass* end)

1238 The class owning (i.e., defining) the property.

1239 • *OverriddenProperty* : Property [0..1] (association *PropertyOverride*)

1240 The property overridden by this property.

1241 • *OverridingProperty* : Property [*] (association *PropertyOverride*)

1242 The property overriding this property.

1243 • *InstanceProperty* : *InstanceProperty* [*] (association *DefiningProperty*)

1244 A value of this property in an instance.

1245 • *OwnedDefaultValue* : Value [0..1] (composition *PropertyDefaultValue*, aggregating on its
1246 *OwningProperty* end)

1247 The default value of the property declaration. A *Value* instance shall be associated if and only if
1248 a default value is defined on the property declaration.

1249 Additional constraints:

1250 1) The value of the *Name* attribute (i.e., the property name) shall follow the formal syntax defined
1251 by the `propertyName` ABNF rule in ANNEX A.

1252 2) Property names shall be unique within its owning (i.e., defining) class.

1253 3) An overriding property shall have the same name as the property it overrides.

1254 Equivalent OCL class constraint:

```

1255 inv: self.PropertyOverride[OverridingProperty]->
1256     size() = 1
1257     implies
1258     self.PropertyOverride[OverridingProperty].
1259     OverriddenProperty.Name.toUpper() =
1260     self.Name.toUpper()

```

1261 NOTE: As a result of constraints 2) and 3), the set of properties exposed by a class may have duplicate
 1262 names if a class defines a property with the same name as a property it inherits without overriding it.

1263 4) The class owning an overridden property shall be a (direct or indirect) superclass of the class
 1264 owning the overriding property.

1265 5) For ordinary properties, the data type of the overriding property shall be the same as the data
 1266 type of the overridden property.

1267 Equivalent OCL class constraint:

```

1268 inv: self.oclIsTypeOf(Meta_Property) and
1269     PropertyOverride[OverridingProperty]->
1270     size() = 1
1271     implies
1272     let pt :Type = /* type of property */
1273     self.ElementType[Element].Type
1274     in
1275     let opt : Type = /* type of overridden prop. */
1276     self.PropertyOverride[OverridingProperty].
1277     OverriddenProperty.Meta_ElementType[Element].Type
1278     in
1279     opt.TypeName.toUpper() = pt.TypeName.toUpper() and
1280     opt.IsArray = pt.IsArray and
1281     opt.ArraySize = pt.ArraySize

```

1282 6) For references, the class referenced by the overriding reference shall be the same as, or a
 1283 subclass of, the class referenced by the overridden reference.

1284 7) A property shall have no more than one initialization constraint defined (either via its default
 1285 value or via the *PropertyConstraint* qualifier, see 5.5.3.39).

1286 8) A property shall have no more than one derivation constraint defined (via the *PropertyConstraint*
 1287 qualifier, see 5.5.3.39).

1288 5.1.2.9 Method

1289 Models a CIM method. A CIM method is the declaration of a behavioral feature of a CIM class,
 1290 representing the ability for invoking an associated behavior.

1291 The CIM data type of the method defines the declared return type of the method.

1292 Methods are inherited to subclasses such that subclasses have the inherited methods in addition to the
 1293 methods defined in the subclass. The combined set of methods defined in a class and methods inherited
 1294 from superclasses is called the methods exposed by the class.

1295 A class defining a method may indicate that the method overrides an inherited method. In this case, the
 1296 class exposes only the overriding method. The characteristics of the overriding method are formed by
 1297 using the characteristics of the overridden method as a basis, changing them as defined in the overriding
 1298 method, within certain limits as defined in section "Additional constraints".

1299 Classes that define a property without overriding an inherited property of the same name, expose two
 1300 properties with that name. This is an undesirable situation since the resolution of property names to the
 1301 actual properties is undefined in this document.

1302 DEPRECATED

1303 Within a single given schema (as defined in 5.1.2.6), the definition of properties without overriding
 1304 inherited properties of the same name defined in a class of the same schema is deprecated. The
 1305 deprecation only applies to the act of establishing that scenario, not necessarily to any schema elements
 1306 that are involved.

1307 DEPRECATED

1308 Between an underlying schema (e.g., the DMTF published CIM schema) and a derived schema (e.g., a
 1309 vendor schema), the definition of properties in the derived schema without overriding inherited properties
 1310 of the same name defined in a class of the underlying schema may occur if both schemas are updated
 1311 independently. Therefore, care should be exercised by the owner of the derived schema when moving to
 1312 a new release of the underlying schema in order to avoid this situation.

1313 Generalization: *TypedElement*

1314 Non-default UML characteristics: None

1315 Attributes: None

1316 Association ends:

1317 • *OwningClass* : Class [1] (composition *MethodDomain*, aggregating on its *OwningClass* end)

1318 The class owning (i.e., defining) the method.

1319 • *OwnedParameter* : Parameter [*] (composition *MethodParameter*, aggregating on its
 1320 *OwningMethod* end)

1321 The parameters of the method. The return value of a method is not represented as a parameter.

1322 • *OverriddenMethod* : Method [0..1] (association *MethodOverride*)

1323 The method overridden by this method.

1324 • *OverridingMethod* : Method [*] (association *MethodOverride*)

1325 The method overriding this method.

1326 Additional constraints:

1327 1) The value of the *Name* attribute (i.e., the method name) shall follow the formal syntax defined
 1328 by the `methodName` ABNF rule in ANNEX A.

1329 2) Method names shall be unique within its owning (i.e., defining) class.

1330 3) An overriding method shall have the same name as the method it overrides.

1331 Equivalent OCL class constraint:

```
1332 inv: self.MethodOverride[OverridingMethod]->
1333     size() = 1
1334     implies
1335         self.MethodOverride[OverridingMethod].
1336             OverriddenMethod.Name.toUpper() =
1337             self.Name.toUpper()
```

1338 NOTE: As a result of constraints 2) and 3), the set of methods exposed by a class may have duplicate
 1339 names if a class defines a method with the same name as a method it inherits without overriding it.

1340 4) The return type of a method shall not be an array.

1341 Equivalent OCL class constraint:

```
1342 inv: self.ElementType[Element].Type.IsArray = false
```

1343 5) The class owning an overridden method shall be a superclass of the class owning the overriding
1344 method.

1345 6) An overriding method shall have the same signature (i.e., parameters and return type) as the
1346 method it overrides.

1347 Equivalent OCL class constraint:

```
1348 inv: MethodOverride[OverridingMethod]->size() = 1
1349     implies
1350         let om : Method = /* overridden method */
1351             self.MethodOverride[OverridingMethod].
1352                 OverriddenMethod
1353         in
1354             om.ElementType[Element].Type.TypeName.toUpper() =
1355                 self.ElementType[Element].Type.TypeName.toUpper()
1356         and
1357         Set {1 .. om.MethodParameter[OwningMethod].
1358             OwnedParameter->size()}
1359         ->forall( i /
1360             let omp : Parameter = /* parm in overridden method */
1361                 om.MethodParameter[OwningMethod].OwnedParameter->
1362                     asOrderedSet()->at(i)
1363             in
1364                 let selfp : Parameter = /* parm in overriding method */
1365                     self.MethodParameter[OwningMethod].OwnedParameter->
1366                         asOrderedSet()->at(i)
1367             in
1368                 omp.Name.toUpper() = selfp.Name.toUpper() and
1369                 omp.ElementType[Element].Type.TypeName.toUpper() =
1370                     selfp.ElementType[Element].Type.TypeName.toUpper()
1371         )
```

1372 5.1.2.10 Parameter

1373 Models a CIM parameter. A CIM parameter is the declaration of a parameter of a CIM method. The return
1374 value of a method is not modeled as a parameter.

1375 Generalization: *TypedElement*

1376 Non-default UML characteristics: None

1377 Attributes: None

1378 Association ends:

- 1379 • *OwningMethod* : *Method* [1] (composition *MethodParameter*, aggregating on its
1380 *OwningMethod* end)

1381 The method owning (i.e., defining) the parameter.

1382 Additional constraints:

- 1383 1) The value of the *Name* attribute (i.e., the parameter name) shall follow the formal syntax defined
1384 by the `parameterName` ABNF rule in ANNEX A.

1385 5.1.2.11 Trigger

1386 Models a CIM trigger. A CIM trigger is the specification of a rule on a CIM element that defines when the
1387 trigger is to be fired.

1388 Triggers may be fired on the following occasions:

- 1389 • On creation, deletion, modification, or access of CIM instances of ordinary classes and
1390 associations. The trigger is specified on the class in this case and applies to all instances.
- 1391 • On modification, or access of a CIM property. The trigger is specified on the property in this
1392 case and applies to all instances.
- 1393 • Before and after the invocation of a CIM method. The trigger is specified on the method in this
1394 case and applies to all invocations of the method.
- 1395 • When a CIM indication is raised. The trigger is specified on the indication in this case and
1396 applies to all occurrences for when this indication is raised.

1397 The rules for when a trigger is to be fired are specified with the *TriggerType* qualifier.

1398 The firing of a trigger shall cause the indications to be raised that are associated to the trigger via
1399 *TriggeredIndication*.

1400 Generalization: *NamedElement*

1401 Non-default UML characteristics: None

1402 Attributes: None

1403 Association ends:

- 1404 • Element : *NamedElement* [1..*] (association *TriggeringElement*)
1405 The CIM element on which the trigger is specified.
- 1406 • Indication : *Indication* [*] (association *TriggeredIndication*)
1407 The CIM indications to be raised when the trigger fires.

1408 Additional constraints:

- 1409 1) The value of the *Name* attribute (i.e., the name of the trigger) shall be unique within the class,
1410 property, or method on which the trigger is specified.
- 1411 2) Triggers shall be specified only on ordinary classes, associations, properties (including
1412 references), methods and indications.

1413 Equivalent OCL class constraint:

```
1414 inv: let e : NamedElement = /* the element on which the trigger is specified*/
1415     self.TriggeringElement[Trigger].Element
1416     in
1417     e.oclIsTypeOf(Class) or
1418     e.oclIsTypeOf(Association) or
1419     e.oclIsTypeOf(Property) or
1420     e.oclIsTypeOf(Reference) or
1421     e.oclIsTypeOf(Method) or
1422     e.oclIsTypeOf(Indication)
```

1423 **5.1.2.12 Indication**

1424 Models a CIM indication. An instance of a CIM indication represents an event that has occurred. If an
 1425 instance of an indication is created, the indication is said to be *raised*. The event causing an indication to
 1426 be raised may be that a trigger has fired, but other arbitrary events may cause an indication to be raised
 1427 as well.

1428 Generalization: *Class*

1429 Non-default UML characteristics: None

1430 Attributes: None

1431 Association ends:

- 1432 • *Trigger*: Trigger [*] (association *TriggeredIndication*)

1433 The triggers that when fired cause the indication to be raised.

1434 Additional constraints:

- 1435 1) An indication shall not own any methods.

1436 Equivalent OCL class constraint:

```
1437 inv: self.MethodDomain[OwningClass].OwnedMethod->size() = 0
```

1438 **5.1.2.13 Association**

1439 Models a CIM association. A CIM association is a special kind of CIM class that represents a relationship
 1440 between two or more CIM classes. A CIM association owns its association ends (i.e., references). This
 1441 allows for adding associations to a schema without affecting the associated classes.

1442 Generalization: *Class*

1443 Non-default UML characteristics: None

1444 Attributes: None

1445 Association ends: None

1446 Additional constraints:

- 1447 1) The superclass of an association shall be an association.

1448 Equivalent OCL class constraint:

```
1449 inv: self.Generalization[SubClass].SuperClass->  
1450 oclIsTypeOf(Association)
```

- 1451 2) An association shall own two or more references.

1452 Equivalent OCL class constraint:

```
1453 inv: self.PropertyDomain[OwningClass].OwnedProperty->  
1454 select( p / p.ocIsTypeOf(Reference) )->size() >= 2
```

- 1455 3) The number of references exposed by an association (i.e., its arity) shall not change in its
1456 subclasses.

1457 Equivalent OCL class constraint:

```
1458 inv: self.PropertyDomain[OwningClass].OwnedProperty->
1459       select( p / p.ocIsTypeOf(Reference) )->size() =
1460       self.Generalization[SubClass].SuperClass->
1461       PropertyDomain[OwningClass].OwnedProperty->
1462       select( p / p.ocIsTypeOf(Reference) )->size()
```

1463 5.1.2.14 Reference

1464 Models a CIM reference. A CIM reference is a special kind of CIM property that represents an association
1465 end, as well as a role the referenced class plays in the context of the association owning the reference.

1466 Generalization: *Property*

1467 Non-default UML characteristics: None

1468 Attributes: None

1469 Association ends: None

1470 Additional constraints:

- 1471 1) The value of the *Name* attribute (i.e., the reference name) shall follow the formal syntax defined
1472 by the `referenceName` ABNF rule in ANNEX A.
- 1473 2) A reference shall be owned by an association (i.e., not by an ordinary class or by an indication).

1474 As a result of this, reference names do not need to be unique within any of the associated
1475 classes.

1476 Equivalent OCL class constraint:

```
1477 inv: self.PropertyDomain[OwnedProperty].OwningClass.  
1478       ocIsTypeOf(Association)
```

1479 5.1.2.15 Qualifier Type

1480 Models the declaration of a CIM qualifier (i.e., a qualifier type). A CIM qualifier is meta data that provides
1481 additional information about the element on which the qualifier is specified.

1482 The qualifier type is either explicitly defined in the CIM namespace, or implicitly defined on an element as
1483 a result of a qualifier that is specified on an element for which no explicit qualifier type is defined.

1484 DEPRECATED

1485 The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

1486 DEPRECATED

1487 Generalization: *TypedElement*

1488 Non-default UML characteristics: None

1489 Attributes:

- 1490 • Scope : string [*]

1491 The scopes of the qualifier. The qualifier scopes determine to which kinds of elements a
1492 qualifier may be specified on. Each qualifier scope shall be one of the following keywords:

- 1493 – "any" - the qualifier may be specified on any qualifiable element.
- 1494 – "class" - the qualifier may be specified on any ordinary class.
- 1495 – "association" - the qualifier may be specified on any association.
- 1496 – "indication" - the qualifier may be specified on any indication.
- 1497 – "property" - the qualifier may be specified on any ordinary property.
- 1498 – "reference" - the qualifier may be specified on any reference.
- 1499 – "method" - the qualifier may be specified on any method.
- 1500 – "parameter" - the qualifier may be specified on any parameter.

1501 Qualifiers cannot be specified on qualifiers.

1502 Association ends:

- 1503 • *Flavor* : *Flavor* [1] (composition *QualifierTypeFlavor*, aggregating on its *QualifierType* end)

1504 The flavor of the qualifier type.

- 1505 • *Qualifier* : *Qualifier* [*] (association *DefiningQualifier*)

1506 The specified qualifiers (i.e., usages) of the qualifier type.

- 1507 • *Element* : *NamedElement* [0..1] (association *ElementQualifierType*)

1508 For implicitly defined qualifier types, the element on which the qualifier type is defined.

1509 **DEPRECATED**

1510 The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

1511 **DEPRECATED**

1512 Qualifier types defined explicitly are not associated to elements; they are global in the CIM namespace.

1513 Additional constraints:

- 1514 1) The value of the *Name* attribute (i.e., the name of the qualifier) shall follow the formal syntax
1515 defined by the `qualifierName` ABNF rule in ANNEX A.

- 1516 2) The names of explicitly defined qualifier types shall be unique within the CIM namespace.

1517 NOTE: Unlike classes, qualifier types are not part of a schema, so name uniqueness cannot be defined at
1518 the definition level relative to a schema, and is instead only defined at the object level relative to a
1519 namespace.

- 1520 3) The names of implicitly defined qualifier types shall be unique within the scope of the CIM
1521 element on which the qualifiers are specified.

- 1522 4) Implicitly defined qualifier types shall agree in data type, scope, flavor and default value with
1523 any explicitly defined qualifier types of the same name.

1524 DEPRECATED

1525 The concept of implicitly defined qualifier types is deprecated. See 5.1.2.16 for details.

1526 DEPRECATED

1527 5.1.2.16 Qualifier

1528 Models the specification (i.e., usage) of a CIM qualifier on an element. A CIM qualifier is meta data that
1529 provides additional information about the element on which the qualifier is specified. The specification of a
1530 qualifier on an element defines a value for the qualifier on that element.

1531 If no explicitly defined qualifier type exists with this name in the CIM namespace, the specification of a
1532 qualifier causes an implicitly defined qualifier type (i.e., a *QualifierType* element) to be created on the
1533 qualified element.

1534 DEPRECATED

1535 The concept of implicitly defined qualifier types is deprecated. Use explicitly defined qualifiers instead.

1536 DEPRECATED

1537 Generalization: *NamedElement*

1538 Non-default UML characteristics: None

1539 Attributes:

- 1540 • *Value* : string [*]

1541 The value of the qualifier, in its string representation.

1542 Association ends:

- 1543 • *QualifierType* : *QualifierType* [1] (association *DefiningQualifier*)

1544 The qualifier type defining the characteristics of the qualifier.

- 1545 • *OwningElement* : *NamedElement* [1] (composition *SpecifiedQualifier*, aggregating on its
1546 *OwningElement* end)

1547 The element on which the qualifier is specified.

1548 Additional constraints:

- 1549 1) The value of the *Name* attribute (i.e., the name of the qualifier) shall follow the formal syntax
1550 defined by the `qualifierName` ABNF rule in ANNEX A.

1551 5.1.2.17 Flavor

1552 The specification of certain characteristics of the qualifier such as its value propagation from the ancestry
1553 of the qualified element, and translatability of the qualifier value.

1554 Generalization: None

1555 Non-default UML characteristics: None

1556 Attributes:

- 1557 • *InheritancePropagation* : boolean

1558 Indicates whether the qualifier value is to be propagated from the ancestry of an element in
1559 case the qualifier is not specified on the element.

- 1560 • *OverridePermission* : boolean

1561 Indicates whether qualifier values propagated to an element may be overridden by the
1562 specification of that qualifier on the element.

- 1563 • *Translatable* : boolean

1564 Indicates whether qualifier value is translatable.

1565 Association ends:

- 1566 • *QualifierType* : *QualifierType* [1] (composition *QualifierTypeFlavor*, aggregating on its
1567 *QualifierType* end)

1568 The qualifier type defining the flavor.

1569 Additional constraints: None

1570 **5.1.2.18 Instance**

1571 Models a CIM instance. A CIM instance is an instance of a CIM class that specifies values for a subset
1572 (including all) of the properties exposed by its defining class.

1573 A CIM instance in a CIM server shall have exactly the properties exposed by its defining class.

1574 A CIM instance cannot redefine the properties or methods exposed by its defining class and cannot have
1575 qualifiers specified.

1576 Generalization: None

1577 Non-default UML characteristics: None

1578 Attributes: None

1579 Association ends:

- 1580 • *OwnedPropertyValue* : *PropertyValue* [*] (composition *SpecifiedProperty*, aggregating on its
1581 *OwningInstance* end)

1582 The property values specified by the instance.

- 1583 • *DefiningClass* : *Class* [1] (association *DefiningClass*)

1584 The defining class of the instance.

1585 Additional constraints:

- 1586 1) A particular property shall be specified at most once in a given instance.

1587 **5.1.2.19 InstanceProperty**

1588 The definition of a property value within a CIM instance.

1589 Generalization: None

1590 Non-default UML characteristics: None

1591 Attributes:

- 1592 • *OwnedValue* : Value [1] (composition *PropertyValue*, aggregating on its
- 1593 *OwningInstanceProperty* end)

1594 The value of the property.

1595 Association ends:

- 1596 • *OwningInstance* : Instance [1] (composition *SpecifiedProperty*, aggregating on its
- 1597 *OwningInstance* end)

1598 The instance for which a property value is defined.

- 1599 • *DefiningProperty* : PropertyValue [1] (association *DefiningProperty*)

1600 The declaration of the property for which a value is defined.

1601 Additional constraints: None

1602 5.1.2.20 Value

1603 A typed value, used in several contexts.

1604 Generalization: None

1605 Non-default UML characteristics: None

1606 Attributes:

- 1607 • *Value* : string [*]

1608 The scalar value or the array of values. Each value is represented as a string.

- 1609 • *IsNull* : boolean

1610 The NULL indicator of the value. If true, the value is NULL. If false, the value is indicated

1611 through the Value attribute.

1612 Association ends:

- 1613 • *OwnedType* : Type [1] (composition *ValueType*, aggregating on its *OwningValue* end)

1614 The type of this value.

- 1615 • *OwningProperty* : Property [0..1] (composition *PropertyDefaultValue*, aggregating on its
- 1616 *OwningProperty* end)

1617 A property declaration that defines this value as its default value.

- 1618 • *OwningInstanceProperty* : InstanceProperty [0..1] (composition *PropertyValue*, aggregating on
- 1619 its *OwningInstanceProperty* end)

1620 A property defined in an instance that has this value.

- 1621 • *OwningQualifierType* : QualifierType [0..1] (composition *QualifierTypeDefaultValue*,
- 1622 aggregating on its *OwningQualifierType* end)

1623 A qualifier type declaration that defines this value as its default value.

- 1624 • *OwningQualifier* : Qualifier [0..1] (composition *QualifierValue*, aggregating on its
- 1625 *OwningQualifier* end)

1626 A qualifier defined on a schema element that has this value.

1627 Additional constraints:

1628 1) If the NULL indicator is set, no values shall be specified.

1629 Equivalent OCL class constraint:

```
1630 inv: self.IsNull = true  
1631     implies self.Value->size() = 0
```

1632 2) If values are specified, the NULL indicator shall not be set.

1633 Equivalent OCL class constraint:

```
1634 inv: self.Value->size() > 0  
1635     implies self.IsNull = false
```

1636 3) A Value instance shall be owned by only one owner.

1637 Equivalent OCL class constraint:

```
1638 inv: self.OwningProperty->size() +  
1639     self.OwningInstanceProperty->size() +  
1640     self.OwningQualifierType->size() +  
1641     self.OwningQualifier->size() = 1
```

1642 5.2 Data Types

1643 Properties, references, parameters, and methods (that is, method return values) have a data type. These
1644 data types are limited to the intrinsic data types or arrays of such. Additional constraints apply to the data
1645 types of some elements, as defined in this document. Structured types are constructed by designing new
1646 classes. There are no subtype relationships among the intrinsic data types uint8, sint8, uint16, sint16,
1647 uint32, sint32, uint64, sint64, string, boolean, real32, real64, datetime, char16, and arrays of them. CIM
1648 elements of any intrinsic data type (including <classname> REF) may have the special value NULL,
1649 indicating absence of value, unless further constrained in this document.

1650 Table 2 lists the intrinsic data types and how they are interpreted.

1651 **Table 2 – Intrinsic Data Types**

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

1652 5.2.1 UCS and Unicode

1653 [ISO/IEC 10646:2003](#) defines the *Universal Multiple-Octet Coded Character Set (UCS)*. [The Unicode](#)
 1654 [Standard](#) defines *Unicode*. This subclause gives a short overview on UCS and Unicode for the scope of
 1655 this document, and defines which of these standards is used by this document.

1656 Even though these two standards define slightly different terminology, they are consistent in the
 1657 overlapping area of their scopes. Particularly, there are matching releases of these two standards that
 1658 define the same UCS/Unicode character repertoire. In addition, each of these standards covers some
 1659 scope that the other does not.

1660 This document uses [ISO/IEC 10646:2003](#) and its terminology. [ISO/IEC 10646:2003](#) references some
 1661 annexes of [The Unicode Standard](#). Where it improves the understanding, this document also states terms
 1662 defined in [The Unicode Standard](#) in parenthesis.

1663 Both standards define two layers of mapping:

1664 *Characters* (Unicode Standard: *abstract characters*) are assigned to UCS *code positions* (Unicode
 1665 Standard: *code points*) in the value space of the integers 0 to 0x10FFFF.

1666 In this document, these code positions are referenced using the U+xxxxxx format defined in [ISO/IEC](#)
 1667 [10646:2003](#). In that format, the aforementioned value space would be stated as U+0000 to
 1668 U+10FFFF.

1669 Not all UCS code positions are assigned to characters; some code positions have a special purpose
 1670 and most code positions are available for future assignment by the standard.

1671 For some characters, there are multiple ways to represent them at the level of code positions. For
1672 example, the character "LATIN SMALL LETTER A WITH GRAVE" (à) can be represented as a
1673 single *precomposed character* at code position U+00E0 (à), or as a sequence of two characters: A
1674 *base character* at code position U+0061 (a), followed by a *combination character* at code position
1675 U+0300 (´). [ISO/IEC 10646:2003](#) references [The Unicode Standard, Version 5.2.0, Annex #15:](#)
1676 [Unicode Normalization Forms](#) for the definition of *normalization forms*. That annex defines four
1677 normalization forms, each of which reduces such multiple ways for representing characters in the
1678 UCS code position space to a single and thus predictable way. The [Character Model for the World](#)
1679 [Wide Web 1.0: Normalization](#) recommends using *Normalization Form C (NFC)* defined in that annex
1680 for all content, because this form avoids potential interoperability problems arising from the use of
1681 canonically equivalent, yet differently represented, character sequences in document formats on the
1682 Web. NFC uses precomposed characters where possible, but not all characters of the UCS
1683 character repertoire can be represented as precomposed characters.

1684 UCS code position values are assigned to binary data values of a certain size that can be stored in
1685 computer memory.

1686 The set of rules governing the assignment of a set of UCS code points to a set of to binary data
1687 values is called a *coded representation form* (Unicode Standard: *encoding form*). Examples are
1688 UCS-2, UTF-16 or UTF-8.

1689 Two sequences of binary data values representing UCS characters that use the same normalization form
1690 and the same coded representation form can be compared for equality of the characters by performing a
1691 binary (e.g., octet-wise) comparison for equality.

1692 5.2.2 String Type

1693 Non-NULL string typed values shall contain zero or more UCS characters (see 5.2.1).

1694 Implementations shall support a character repertoire for string typed values that is that defined by
1695 [ISO/IEC 10646:2003](#) with its amendments [ISO/IEC 10646:2003/Amd 1:2005](#) and [ISO/IEC](#)
1696 [10646:2003/Amd 2:2006](#) applied (this is the same character repertoire as defined by the Unicode
1697 Standard 5.0).

1698 It is recommended that implementations support the latest published UCS character repertoire in a timely
1699 manner.

1700 UCS characters in string typed values should be represented in Normalization Form C (NFC), as defined
1701 in [The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms](#).

1702 UCS characters in string typed values shall be represented in a coded representation form that satisfies
1703 the requirements for the character repertoire stated in this subclause. Other specifications are expected
1704 to specify additional rules on the usage of particular coded representation forms (see [DSP0200](#) as an
1705 example). In order to minimize the need for any conversions between different coded representation
1706 forms, it is recommended that such other specifications mandate the UTF-8 coded representation form
1707 (defined in [ISO/IEC 10646:2003](#)).

1708 NOTE: Version 2.6.0 of this document introduced the requirement to support at least the character repertoire of
1709 [ISO/IEC 10646:2003](#) with its amendments [ISO/IEC 10646:2003/Amd 1:2005](#) and [ISO/IEC 10646:2003/Amd](#)
1710 [2:2006](#) applied. Previous versions of this document simply stated that the string type is a "UCS-2 string" without
1711 offering further details as to whether this was a definition of the character repertoire or a requirement on the usage of
1712 that coded representation form. UCS-2 does not support the character repertoire required in this subclause, and it
1713 does not satisfy the requirements of a number of countries, including the requirements of the Chinese national
1714 standard GB18030. UCS-2 was superseded by UTF-16 in Unicode 2.0 (released in 1996), although it is still in use
1715 today. For example, CIM clients that still use UCS-2 as an internal representation of string typed values will not be
1716 able to represent all characters that may be returned by a CIM server that supports the character repertoire required
1717 in this subclause.

1718 5.2.3 Char16 Type

1719 The char16 type is a 16-bit data entity. Non-NULL char16 typed values shall contain one UCS character
1720 (see 5.2.1) in the coded representation form UCS-2 (defined in [ISO/IEC 10646:2003](#)).

1721 DEPRECATED

1722 Due to the limitations of UCS-2 (see 5.2.2), the char16 type is deprecated since version 2.6.0 of this
1723 document. Use the string type instead.

1724 DEPRECATED

1725 5.2.4 Datetime Type

1726 The datetime type specifies a timestamp (point in time) or an interval. If it specifies a timestamp, the
1727 timezone offset can be preserved. In both cases, datetime specifies the date and time information with
1728 varying precision.

1729 Datetime uses a fixed string-based format. The format for timestamps is:

1730 `yyyymmddhhmmss.mmmmmmsutc`

1731 The meaning of each field is as follows:

- 1732 • `yyyy` is a 4-digit year.
- 1733 • `mm` is the month within the year (starting with 01).
- 1734 • `dd` is the day within the month (starting with 01).
- 1735 • `hh` is the hour within the day (24-hour clock, starting with 00).
- 1736 • `mm` is the minute within the hour (starting with 00).
- 1737 • `ss` is the second within the minute (starting with 00).
- 1738 • `mmmmmm` is the microsecond within the second (starting with 000000).
- 1739 • `s` is a + (plus) or – (minus), indicating that the value is a timestamp with the sign of Universal
1740 Coordinated Time (UTC), which is basically the same as Greenwich Mean Time correction field.
1741 A + (plus) is used for time zones east of Greenwich, and a – (minus) is used for time zones
1742 west of Greenwich.
- 1743 • `utc` is the offset from UTC in minutes (using the sign indicated by `s`).

1744 Timestamps are based on the proleptic Gregorian calendar, as defined in section 3.2.1, "The Gregorian
1745 calendar", of [ISO 8601:2004](#).

1746 Because datetime contains the time zone information, the original time zone can be reconstructed from
1747 the value. Therefore, the same timestamp can be specified using different UTC offsets by adjusting the
1748 hour and minutes fields accordingly.

1749 For example, Monday, May 25, 1998, at 1:30:15 PM EST is represented as

1750 `19980525133015.0000000-300`.

1751 An alternative representation of the same timestamp is `19980525183015.0000000+000`.

1752 The format for intervals is as follows:

1753 `dddddddhmmss.mmmmm:000`

1754 The meaning of each field is as follows:

- 1755 • `ddddddddd` is the number of days.
- 1756 • `hh` is the remaining number of hours.
- 1757 • `mm` is the remaining number of minutes.
- 1758 • `ss` is the remaining number of seconds.
- 1759 • `mmmmmmm` is the remaining number of microseconds.
- 1760 • `:` (colon) indicates that the value is an interval.
- 1761 • `000` (the UTC offset field) is always zero for interval properties.

1762 For example, an interval of 1 day, 13 hours, 23 minutes, 12 seconds, and 0 microseconds would be
1763 represented as follows:

1764 `00000001132312.000000:000`

1765 For both timestamps and intervals, the field values shall be zero-padded so that the entire string is always
1766 25 characters in length.

1767 For both timestamps and intervals, fields that are not significant shall be replaced with the asterisk (*)
1768 character. Fields that are not significant are beyond the resolution of the data source. These fields
1769 indicate the precision of the value and can be used only for an adjacent set of fields, starting with the
1770 least significant field (mmmmmm) and continuing to more significant fields. The granularity for asterisks is
1771 always the entire field, except for the mmmmmm field, for which the granularity is single digits. The UTC
1772 offset field shall not contain asterisks.

1773 For example, if an interval of 1 day, 13 hours, 23 minutes, 12 seconds, and 125 milliseconds is measured
1774 with a precision of 1 millisecond, the format is: `00000001132312.125***:000`.

1775 The following operations are defined on datetime types:

- 1776 • Arithmetic operations:
 - 1777 – Adding or subtracting an interval to or from an interval results in an interval.
 - 1778 – Adding or subtracting an interval to or from a timestamp results in a timestamp.
 - 1779 – Subtracting a timestamp from a timestamp results in an interval.
 - 1780 – Multiplying an interval by a numeric or vice versa results in an interval.
 - 1781 – Dividing an interval by a numeric results in an interval.
 - 1782 Other arithmetic operations are not defined.
- 1783 • Comparison operations:
 - 1784 – Testing for equality of two timestamps or two intervals results in a boolean value.
 - 1785 – Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in
1786 a boolean value.
 - 1787 Other comparison operations are not defined.
 - 1788 Comparison between a timestamp and an interval and vice versa is not defined.

1789 Specifications that use the definition of these operations (such as specifications for query languages)
1790 should state how undefined operations are handled.

- 1791 Any operations on datetime types in an expression shall be handled as if the following sequential steps
1792 were performed:
- 1793 1) Each datetime value is converted into a range of microsecond values, as follows:
- 1794 • The lower bound of the range is calculated from the datetime value, with any asterisks
1795 replaced by their minimum value.
 - 1796 • The upper bound of the range is calculated from the datetime value, with any asterisks
1797 replaced by their maximum value.
 - 1798 • The basis value for timestamps is the oldest valid value (that is, 0 microseconds
1799 corresponds to 00:00.000000 in the timezone with datetime offset +720, on January 1 in
1800 the year 1 BCE, using the proleptic Gregorian calendar). This definition implicitly performs
1801 timestamp normalization.
- 1802 NOTE: 1 BCE is the year before 1 CE.
- 1803 2) The expression is evaluated using the following rules for any datetime ranges:
- 1804 • Definitions:
 - 1805 T(x, y) The microsecond range for a timestamp with the lower bound x and the upper
1806 bound y
 - 1807 I(x, y) The microsecond range for an interval with the lower bound x and the upper
1808 bound y
 - 1809 D(x, y) The microsecond range for a datetime (timestamp or interval) with the lower
1810 bound x and the upper bound y
 - 1811 • Rules:
 - 1812 $I(a, b) + I(c, d) := I(a+c, b+d)$
 - 1813 $I(a, b) - I(c, d) := I(a-d, b-c)$
 - 1814 $T(a, b) + I(c, d) := T(a+c, b+d)$
 - 1815 $T(a, b) - I(c, d) := T(a-d, b-c)$
 - 1816 $T(a, b) - T(c, d) := I(a-d, b-c)$
 - 1817 $I(a, b) * c := I(a*c, b*c)$
 - 1818 $I(a, b) / c := I(a/c, b/c)$
 - 1819 $D(a, b) < D(c, d) := \text{true if } b < c, \text{ false if } a \geq d, \text{ otherwise NULL (uncertain)}$
 - 1820 $D(a, b) \leq D(c, d) := \text{true if } b \leq c, \text{ false if } a > d, \text{ otherwise NULL (uncertain)}$
 - 1821 $D(a, b) > D(c, d) := \text{true if } a > d, \text{ false if } b \leq c, \text{ otherwise NULL (uncertain)}$
 - 1822 $D(a, b) \geq D(c, d) := \text{true if } a \geq d, \text{ false if } b < c, \text{ otherwise NULL (uncertain)}$
 - 1823 $D(a, b) = D(c, d) := \text{true if } a = b = c = d, \text{ false if } b < c \text{ OR } a > d, \text{ otherwise NULL}$
1824 (uncertain)
 - 1825 $D(a, b) \neq D(c, d) := \text{true if } b < c \text{ OR } a > d, \text{ false if } a = b = c = d, \text{ otherwise NULL}$
1826 (uncertain)
- 1827 These rules follow the well-known mathematical interval arithmetic. For a definition of
1828 mathematical interval arithmetic, see http://en.wikipedia.org/wiki/Interval_arithmetic.
- 1829 NOTE 1: Mathematical interval arithmetic is commutative and associative for addition and
1830 multiplication, as in ordinary arithmetic.
- 1831 NOTE 2: Mathematical interval arithmetic mandates the use of three-state logic for the result of
1832 comparison operations. A special value called "uncertain" indicates that a decision cannot be made.
1833 The special value of "uncertain" is mapped to the NULL value in datetime comparison operations.

1834 3) Overflow and underflow condition checking is performed on the result of the expression, as
1835 follows:

1836 For timestamp results:

- 1837 • A timestamp older than the oldest valid value in the timezone of the result produces
1838 an arithmetic underflow condition.
- 1839 • A timestamp newer than the newest valid value in the timezone of the result produces
1840 an arithmetic overflow condition.

1841 For interval results:

- 1842 • A negative interval produces an arithmetic underflow condition.
- 1843 • A positive interval greater than the largest valid value produces an arithmetic overflow
1844 condition.

1845 Specifications using these operations (for instance, query languages) should define how these
1846 conditions are handled.

1847 4) If the result of the expression is a datetime type, the microsecond range is converted into a valid
1848 datetime value such that the set of asterisks (if any) determines a range that matches the actual
1849 result range or encloses it as closely as possible. The GMT timezone shall be used for any
1850 timestamp results.

1851 NOTE: For most fields, asterisks can be used only with the granularity of the entire field.

1852 Examples:

```
1853 "20051003110000.000000+000" + "00000000002233.000000:000"
1854     evaluates to "20051003112233.000000+000"
1855
1856 "20051003110000.*****+000" + "00000000002233.000000:000"
1857     evaluates to "20051003112233.*****+000"
1858
1859 "20051003110000.*****+000" + "00000000002233.00000*:000"
1860     evaluates to "200510031122**.******+000"
1861
1862 "20051003110000.*****+000" + "00000000002233.*****:000"
1863     evaluates to "200510031122**.******+000"
1864
1865 "20051003110000.*****+000" + "00000000005959.*****:000"
1866     evaluates to "20051003*****.******+000"
1867
1868 "20051003110000.*****+000" + "000000000022**.******:000"
1869     evaluates to "2005100311****.******+000"
1870
1871 "20051003112233.000000+000" - "00000000002233.000000:000"
1872     evaluates to "20051003110000.000000+000"
1873
1874 "20051003112233.*****+000" - "00000000002233.000000:000"
1875     evaluates to "20051003110000.******+000"
1876
1877 "20051003112233.*****+000" - "00000000002233.00000*:000"
1878     evaluates to "20051003110000.******+000"
1879
1880 "20051003112233.*****+000" - "00000000002232.*****:000"
1881     evaluates to "200510031100**.******+000"
1882
1883 "20051003112233.*****+000" - "00000000002233.*****:000"
1884     evaluates to "20051003*****.******+000"
1885
1886 "20051003060000.000000-300" + "00000000002233.000000:000"
```

```

1887     evaluates to "20051003112233.000000+000"
1888
1889 "20051003060000.*****-300" + "00000000002233.000000:000"
1890     evaluates to "20051003112233.*****+000"
1891
1892 "000000000011**.*****:000" * 60
1893     evaluates to "0000000011****.*****:000"
1894
1895 60 times adding up "000000000011**.*****:000"
1896     evaluates to "0000000011****.*****:000"
1897
1898 "20051003112233.000000+000" = "20051003112233.000000+000"
1899     evaluates to true
1900
1901 "20051003122233.000000+060" = "20051003112233.000000+000"
1902     evaluates to true
1903
1904 "20051003112233.*****+000" = "20051003112233.*****+000"
1905     evaluates to NULL (uncertain)
1906
1907 "20051003112233.*****+000" = "200510031122**.*****+000"
1908     evaluates to NULL (uncertain)
1909
1910 "20051003112233.*****+000" = "20051003112234.*****+000"
1911     evaluates to false
1912
1913 "20051003112233.*****+000" < "20051003112234.*****+000"
1914     evaluates to true
1915
1916 "20051003112233.5*****+000" < "20051003112233.*****+000"
1917     evaluates to NULL (uncertain)

```

1918 A datetime value is valid if the value of each single field is in the valid range. Valid values shall not be
 1919 rejected by any validity checking within the CIM infrastructure.

1920 Within these valid ranges, some values are defined as reserved. Values from these reserved ranges shall
 1921 not be interpreted as points in time or durations.

1922 Within these reserved ranges, some values have special meaning. The CIM schema should not define
 1923 additional class-specific special values from the reserved range.

1924 The valid and reserved ranges and the special values are defined as follows:

- 1925 • For timestamp values:

1926 Oldest valid timestamp: "00000101000000.000000+720"

1927 Reserved range (1 million values)

1928 Oldest useable timestamp: "00000101000001.000000+720"

1929 Range interpreted as points in time

1930 Youngest useable timestamp: "99991231115959.999998-720"

1931 Reserved range (1 value)

1932 Youngest valid timestamp: "99991231115959.999999-720"

1933 Special values in the reserved ranges:

1934 "Now": "00000101000000.000000+720"

1935	"Infinite past":	"00000101000000.999999+720"
1936	"Infinite future":	"99991231115959.999999-720"
1937	• For interval values:	
1938	Smallest valid and useable interval:	"00000000000000.000000:000"
1939		Range interpreted as durations
1940	Largest useable interval:	"99999999235958.999999:000"
1941		Reserved range (1 million values)
1942	Largest valid interval:	"99999999235959.999999:000"
1943	Special values in reserved range:	
1944	"Infinite duration":	"99999999235959.000000:000"

1945 5.2.5 Indicating Additional Type Semantics with Qualifiers

1946 Because counter and gauge types are actually simple integers with specific semantics, they are not
 1947 treated as separate intrinsic types. Instead, qualifiers must be used to indicate such semantics when
 1948 properties are declared. The following example merely suggests how this can be done; the qualifier
 1949 names chosen are not part of this standard:

```

1950 class ACME_Example
1951 {
1952     [Counter]
1953     uint32 NumberOfCycles;
1954
1955     [Gauge]
1956     uint32 MaxTemperature;
1957
1958     [OctetString, ArrayType("Indexed")]
1959     uint8 IPAddress[10];
1960 };

```

1961 For documentation purposes, implementers are permitted to introduce such arbitrary qualifiers. The
 1962 semantics are not enforced.

1963 5.2.6 Comparison of Values

1964 This subclause defines comparison of values for equality and ordering.

1965 Values of boolean datatypes shall be compared for equality and ordering as if "true" was 1 and "false"
 1966 was 0 and the mathematical comparison rules for integer numbers were used on those values.

1967 Values of integer number datatypes shall be compared for equality and ordering according to the
 1968 mathematical comparison rules for the integer numbers they represent.

1969 Values of real number datatypes shall be compared for equality and ordering according to the rules
 1970 defined in [ANSI/IEEE 754-1985](#).

- 1971 Values of the string and char16 datatypes shall be compared for equality on a UCS character basis, by
 1972 using the string identity matching rules defined in chapter 4 "String Identity Matching" of the [Character](#)
 1973 [Model for the World Wide Web 1.0: Normalization](#) specification. As a result, comparisons between a
 1974 char16 typed value and a string typed value are valid.
- 1975 In order to minimize the processing involved in UCS normalization, string and char16 typed values should
 1976 be stored and transmitted in Normalization Form C (NFC, see 5.2.2) where possible, which allows
 1977 skipping the costly normalization when comparing the strings.
- 1978 This document does not define an order between values of the string and char16 datatypes, since UCS
 1979 ordering rules may be compute intensive and their usage should be decided on a case by case basis.
 1980 The ordering of the "Common Template Table" defined in [ISO/IEC 14651:2007](#) provides a reasonable
 1981 default ordering of UCS strings for human consumption. However, an ordering based on the UCS code
 1982 positions, or even based on the octets of a particular UCS coded representation form is typically less
 1983 compute intensive and may be sufficient, for example when no human consumption of the ordering result
 1984 is needed.
- 1985 Values of schema elements qualified as octetstrings shall be compared for equality and ordering based
 1986 on the sequence of octets they represent. As a result, comparisons across different octetstring
 1987 representations (as defined in 5.5.3.35) are valid. Two sequences of octets shall be considered equal if
 1988 they contain the same number of octets and have equal octets in each octet pair in the sequences. An
 1989 octet sequence S1 shall be considered less than an octet sequence S2, if the first pair of different octets,
 1990 reading from left to right, is beyond the end of S1 or has an octet in S1 that is less than the octet in S2.
 1991 This comparison rule yields the same results as the comparison rule defined for the strcmp() function in
 1992 [IEEE Std 1003.1, 2004 Edition](#).
- 1993 Two values of the reference datatype shall be considered equal if they resolve to the same CIM object in
 1994 the same namespace. This document does not define an order between two values of the reference
 1995 datatype.
- 1996 Two values of the datetime datatype shall be compared based on the time duration or point in time they
 1997 represent, according to mathematical comparison rules for these numbers. As a result, two datetime
 1998 values that represent the same point in time using different timezone offsets are considered equal.
- 1999 Two values of compatible datatypes that both are NULL shall be considered equal. This document does
 2000 not define an order between two values of compatible datatypes where one is NULL, and the other is not
 2001 NULL.
- 2002 Two array values of compatible datatypes shall be considered equal if they contain the same number of
 2003 array entries and in each pair of array entries, the two array entries are equal. This document does not
 2004 define an order between two array values.

2005 **5.3 Supported Schema Modifications**

- 2006 This subclause lists typical modifications of schema definitions and qualifier type declarations and defines
 2007 their compatibility. Such modifications might be introduced into an existing CIM environment by upgrading
 2008 the schema to a newer schema version. However, any rules for the modification of schema related
 2009 objects (i.e., classes and qualifier types) in a CIM server are outside of the scope of this document.
 2010 Specifications dealing with modification of schema related objects in a CIM server should define such
 2011 rules and should consider the compatibility defined in this subclause.
- 2012 Table 3 lists modifications of an existing schema definition (including an empty schema). The compatibility
 2013 of the modification is indicated for CIM clients that utilize the modified element, and for a CIM server that
 2014 implements the modified element. Compatibility for a CIM server that utilizes the modified element (e.g.,
 2015 via so called "up-calls") is the same as for a CIM client that utilizes the modified element.
- 2016 The compatibility for CIM clients as expressed in Table 3 assumes that the CIM client remains unchanged
 2017 and is exposed to a CIM server that was updated to fully reflect the schema modification.

- 2018 The compatibility for CIM servers as expressed in Table 3 assumes that the CIM server remains
2019 unchanged but is exposed to the modified schema that is loaded into the CIM namespace being serviced
2020 by the CIM server.
- 2021 Compatibility is stated as follows:
- 2022 • Transparent – the respective component does not need to be changed in order to properly deal
2023 with the modification
 - 2024 • Not transparent – the respective component needs to be changed in order to properly deal with
2025 the modification
- 2026 Schema modifications qualified as transparent for both CIM clients and CIM servers are allowed in a
2027 minor version update of the schema. Any other schema modifications are allowed only in a major version
2028 update of the schema.
- 2029 The schema modifications listed in Table 3 cover simple cases, which may be combined to yield more
2030 complex cases. For example, a typical schema change is to move existing properties or methods into a
2031 new superclass. The compatibility of this complex schema modification can be determined by
2032 concatenating simple schema modifications listed in Table 3, as follows:
- 2033 1) SM1: Adding a class to the schema:
2034 The new superclass gets added as an empty class with (yet) no superclass
 - 2035 2) SM3: Inserting an existing class that defines no properties or methods into an inheritance
2036 hierarchy of existing classes:
2037 The new superclass gets inserted into an inheritance hierarchy
 - 2038 3) SM8: Moving an existing property from a class to one of its superclasses (zero or more times)
2039 Properties get moved to the newly inserted superclass
 - 2040 4) SM12: Moving a method from a class to one of its superclasses (zero or more times)
2041 Methods get moved to the newly inserted superclass
- 2042 The resulting compatibility of this complex schema modification for CIM clients is transparent, since all
2043 these schema modifications are transparent. Similarly, the resulting compatibility for CIM servers is
2044 transparent for the same reason.
- 2045 Some schema modifications cause other changes in the schema to happen. For example, the removal of
2046 a class causes any associations or method parameters that reference that class to be updated in some
2047 way.

Table 3 – Compatibility of Schema Modifications

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

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

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

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

Schema Modification	Compatibility for CIM clients	Compatibility for CIM servers	Allowed in a Minor Version Update of the Schema
SM24: Changing the class referenced by a method output parameter or method return value of reference type to a superclass of the previously referenced class	Not Transparent	Transparent	No
SM25: Changing the class referenced by a method output parameter or method return value of reference type to any class other than a subclass or superclass of the previously referenced class	Not Transparent	Not Transparent	No
SM26: Changing a class between ordinary class, association or indication	Not transparent	Not transparent	No
SM27: Reducing or increasing the arity of an association (i.e., increasing or decreasing the number of references exposed by the association)	Not transparent	Not transparent	No
SM28: Changing the effective value of a qualifier on an existing schema element	As defined in the qualifier description in 5.5	As defined in the qualifier description in 5.5	Yes, if transparent for both CIM clients and CIM servers, otherwise No

- 2049 1) Factory approach to populate the properties of any instances to be returned:
- 2050 Some CIM server architectures (e.g., CMPI-based CIM providers) support factory methods that
- 2051 create an internal representation of a CIM instance by inspecting the class object and creating
- 2052 property values for all properties exposed by the class and setting those values to their class
- 2053 defined default values. This delegates the knowledge about newly added properties to the
- 2054 schema definition of the class and will return instances that are compliant to the modified
- 2055 schema without changing the code of the CIM server. A subsequent release of the CIM server
- 2056 can then start supporting the new property with more reasonable values than the class defined
- 2057 default value.
- 2058 Table 4 lists modifications of qualifier types. The compatibility of the modification is indicated for an
- 2059 existing schema. Compatibility for CIM clients or CIM servers is determined by Table 4 (in any
- 2060 modifications that are related to qualifier values).
- 2061 The compatibility for a schema as expressed in Table 4 assumes that the schema remains unchanged
- 2062 but is confronted with a qualifier type declaration that reflects the modification.

2063 Compatibility is stated as follows:

- 2064 • Transparent – the schema does not need to be changed in order to properly deal with the
2065 modification
- 2066 • Not transparent – the schema needs to be changed in order to properly deal with the
2067 modification

2068 CIM supports extension schemas, so the actual usage of qualifiers in such schemas is by definition
2069 unknown and any possible usage needs to be assumed for compatibility considerations.

2070 **Table 4 – Compatibility of Qualifier Type Modifications**

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

2071 5.3.1 Schema Versions

2072 Schema versioning is described in [DSP4004](#). Versioning takes the form m.n.u, where:

- 2073 • m = major version identifier in numeric form
- 2074 • n = minor version identifier in numeric form
- 2075 • u = update (errata or coordination changes) in numeric form

2076 The usage rules for the Version qualifier in 5.5.3.53 provide additional information.

2077 Classes are versioned in the CIM schemas. The Version qualifier for a class indicates the schema release
2078 of the last change to the class. Class versions in turn dictate the schema version. A major version change
2079 for a class requires the major version number of the schema release to be incremented. All class versions
2080 must be at the same level or a higher level than the schema release because classes and models that
2081 differ in minor version numbers shall be backwards-compatible. In other words, valid instances shall
2082 continue to be valid if the minor version number is incremented. Classes and models that differ in major
2083 version numbers are not backwards-compatible. Therefore, the major version number of the schema
2084 release shall be incremented.

2085 Table 5 lists modifications to the CIM schemas in final status that cause a major version number change.
2086 Preliminary models are allowed to evolve based on implementation experience. These modifications
2087 change application behavior and/or customer code. Therefore, they force a major version update and are
2088 discouraged. Table 5 is an exhaustive list of the possible modifications based on current CIM experience
2089 and knowledge. Items could be added as new issues are raised and CIM standards evolve.

2090 Alterations beyond those listed in Table 5 are considered interface-preserving and require the minor
2091 version number to be incremented. Updates/errata are not classified as major or minor in their impact, but
2092 they are required to correct errors or to coordinate across standards bodies.

2093

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

Description	Explanation or Exceptions
Class deletion	
Property deletion or data type change	
Method deletion or signature change	
Reorganization of values in an enumeration	The semantics and mappings of an enumeration cannot change, but values can be added in unused ranges as a minor change or update.
Movement of a class upwards in the inheritance hierarchy; that is, the removal of superclasses from the inheritance hierarchy	The removal of superclasses deletes properties or methods. New classes can be inserted as superclasses as a minor change or update. Inserted classes shall not change keys or add required properties.
Addition of Abstract, Indication, or Association qualifiers to an existing class	
Change of an association reference downward in the object hierarchy to a subclass or to a different part of the hierarchy	The change of an association reference to a subclass can invalidate existing instances.
Addition or removal of a Key or Weak qualifier	
Addition of the Required qualifier to a method input parameter or a property that may be written	<p>Changing to require a non-NULL value to be passed to an input parameter or to be written to a property may break existing CIM clients that pass NULL under the prior definition.</p> <p>An addition of the Required qualifier to method output parameters, method return values and properties that may only be read is considered a compatible change, as CIM clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the CIM server, as defined in 5.5.3.42.</p> <p>The description of an existing schema element that added the Required qualifier in a revision of the schema should indicate the schema version in which this change was made, as defined in 5.5.3.42.</p>
Removal of the Required qualifier from a method output parameter, a method (i.e., its return value) or a property that may be read	<p>Changing to no longer guarantee a non-NULL value to be returned by an output parameter, a method return value, or a property that may be read may break existing CIM clients that relied on the prior guarantee.</p> <p>A removal of the Required qualifier from method input parameters and properties that may only be written is a compatible change, as CIM clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the CIM server, as defined in 5.5.3.42.</p> <p>The description of an existing schema element that removed the Required qualifier in a revision of the schema should indicate the schema version in which this change was made, as defined in 5.5.3.42.</p>
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	

Description	Explanation or Exceptions
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	

2094 5.4 Class Names

2095 Fully-qualified class names are in the form <schema name>_<class name>. An underscore is used as a
2096 delimiter between the <schema name> and the <class name>. The delimiter cannot appear in the
2097 <schema name> although it is permitted in the <class name>.

2098 The format of the fully-qualified name allows the scope of class names to be limited to a schema. That is,
2099 the schema name is assumed to be unique, and the class name is required to be unique only within the
2100 schema. The isolation of the schema name using the underscore character allows user interfaces
2101 conveniently to strip off the schema when the schema is implied by the context.

2102 The following are examples of fully-qualified class names:

- 2103 • CIM_ManagedSystemElement: the root of the CIM managed system element hierarchy
- 2104 • CIM_ComputerSystem: the object representing computer systems in the CIM schema
- 2105 • CIM_SystemComponent: the association relating systems to their components
- 2106 • Win32_ComputerSystem: the object representing computer systems in the Win32 schema

2107 5.5 Qualifiers

2108 Qualifiers are named and typed values that provide information about CIM elements. Since the qualifier
2109 values are on CIM elements and not on CIM instances, they are considered to be meta-data.

2110 Subclause 5.5.1 describes the concept of qualifiers, independently of their representation in MOF. For
2111 their representation in MOF, see 7.7.

2112 Subclauses 5.5.2, 5.5.3, and 5.5.4 describe the meta, standard, and optional qualifiers, respectively. Any
2113 qualifier type declarations with the names of these qualifiers shall have the name, type, scope, flavor, and
2114 default value defined in these subclauses.

2115 Subclause 5.5.5 describes user-defined qualifiers.

2116 Subclause 5.5.6 describes how the MappingString qualifier can be used to define mappings between CIM
2117 and other information models.

2118 5.5.1 Qualifier Concept

2119 5.5.1.1 Qualifier Value

2120 Any qualifiable CIM element (i.e., classes including associations and indications, properties including
2121 references, methods and parameters) shall have a particular set of qualifier values, as follows. A qualifier
2122 shall have a value on a CIM element if that kind of CIM element is in the scope of the qualifier, as defined
2123 in 5.5.1.3. If a kind of CIM element is in the scope of a qualifier, the qualifier is said to be an applicable
2124 qualifier for that kind of CIM element and for a specific CIM element of that kind.

2125 Any applicable qualifier may be specified on a CIM element. When an applicable qualifier is specified on
 2126 a CIM element, the qualifier shall have an explicit value on that CIM element. When an applicable
 2127 qualifier is not specified on a CIM element, the qualifier shall have an assumed value on that CIM
 2128 element, as defined in 5.5.1.5.

2129 The value specified for a qualifier shall be consistent with the data type defined by its qualifier type.

2130 There shall not be more than one qualifier with the same name specified on any CIM element.

2131 **5.5.1.2 Qualifier Type**

2132 A qualifier type defines name, data type, scope, flavor and default value of a qualifier, as follows:

2133 The name of a qualifier is a string that shall follow the formal syntax defined by the `qualifierName`

2134 ABNF rule in ANNEX A.

2135 The data type of a qualifier shall be one of the intrinsic data types defined in Table 2, including arrays of
 2136 such, excluding references and arrays thereof. If the data type is an array type, the array shall be an
 2137 indexed variable length array, as defined in 7.8.2.

2138 The scope of a qualifier determines which kinds of CIM elements have a value of that qualifier, as defined
 2139 in 5.5.1.3.

2140 The flavor of a qualifier determines propagation to subclasses, override permissions, and translatability,
 2141 as defined in 5.5.1.4.

2142 The default value of a qualifier is used to determine the effective value of qualifiers that are not specified
 2143 on a CIM element, as defined in 5.5.1.5.

2144 There shall not exist more than one qualifier type object with the same name in a CIM namespace.
 2145 Qualifier types are not part of a schema; therefore name uniqueness of qualifiers cannot be defined within
 2146 the boundaries of a schema (like it is done for class names).

2147 **5.5.1.3 Qualifier Scope**

2148 The scope of a qualifier determines which kinds of CIM elements have a value for that qualifier.

2149 The scope of a qualifier shall be one or more of the scopes defined in Table 6, except for scope (Any)
 2150 whose specification shall not be combined with the specification of the other scopes. Qualifiers cannot be
 2151 specified on qualifiers.

2152 **Table 6 – Defined Qualifier Scopes**

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

2153 5.5.1.4 Qualifier Flavor

2154 The flavor of a qualifier determines propagation of its value to subclasses, override permissions of the
2155 propagated value, and translatability of the value.

2156 The flavor of a qualifier shall be zero or more of the flavors defined in Table 7, subject to further
2157 restrictions defined in this subclause.

2158 **Table 7 – Defined Qualifier Flavors**

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

2159 Flavor (ToSubclass) and flavor (Restricted) shall not be specified both on the same qualifier type. If none
2160 of these two flavors is specified on a qualifier type, flavor (ToSubclass) shall be the implied default.

2161 If flavor (Restricted) is specified, override permission is meaningless. Thus, flavor (EnableOverride) and
2162 flavor (DisableOverride) should not be specified and are meaningless if specified.

2163 Flavor (EnableOverride) and flavor (DisableOverride) shall not be specified both on the same qualifier
2164 type. If none of these two flavors is specified on a qualifier type, flavor (EnableOverride) shall be the
2165 implied default.

2166 This results in three meaningful combinations of these flavors:

- 2167 • Restricted – propagation to subclasses is disabled
- 2168 • EnableOverride – propagation to subclasses is enabled and override permission is granted
- 2169 • DisableOverride – propagation to subclasses is enabled and override permission is not granted

2170 If override permission is not granted for a qualifier type, then for a particular CIM element in the scope of
2171 that qualifier type, a qualifier with that name may be specified multiple times in the ancestry of its class,
2172 but each occurrence shall specify the same value. This semantics allows the qualifier to change its
2173 effective value at most once along the ancestry of an element.

2174 If flavor (Translatable) is specified on a qualifier type, the specification of localized qualifiers shall be
2175 enabled for that qualifier, otherwise it shall be disabled. Flavor (Translatable) shall be specified only on
2176 qualifier types that have data type string or array of strings. For details, see 5.5.1.6.

2177 5.5.1.5 Effective Qualifier Values

2178 When there is a qualifier type defined for a qualifier, and the qualifier is applicable but not specified on a
2179 CIM element, the CIM element shall have an assumed value for that qualifier. This assumed value is
2180 called the effective value of the qualifier.

2181 The effective value of a particular qualifier on a given CIM element shall be determined as follows:

2182 If the qualifier is specified on the element, the effective value is the value of the specified qualifier. In
2183 MOF, qualifiers may be specified without specifying a value, in which case a value is implied, as
2184 described in 7.7.

- 2185 If the qualifier is not specified on the element and propagation to subclasses is disabled, the effective
2186 value is the default value defined on the qualifier type declaration.
- 2187 If the qualifier is not specified on the element and propagation to subclasses is enabled, the effective
2188 value is the value of the nearest like-named qualifier that is specified in the ancestry of the element. If the
2189 qualifier is not specified anywhere in the ancestry of the element, the effective value is the default value
2190 defined on the qualifier type declaration.
- 2191 The ancestry of an element is the set of elements that results from recursively determining its ancestor
2192 elements. An element is not considered part of its ancestry.
- 2193 The ancestor of an element depends on the kind of element, as follows:
- 2194 • For a class, its superclass is its ancestor element. If the class does not have a superclass, it has
2195 no ancestor.
 - 2196 • For a property (including references) or method, the overridden element is its ancestor. If the
2197 element is not overriding another element, it does not have an ancestor.
 - 2198 • For a parameter of a method, the like-named parameter of the overridden method is its
2199 ancestor. If the method is not overriding another method, its parameters do not have an
2200 ancestor.

2201 5.5.1.6 Localized Qualifiers

2202 Localized qualifiers allow the specification of qualifier values in a specific language.

2203 DEPRECATED

2204 Localized qualifiers and the flavor (Translatable) as described in this subclause have been deprecated.
2205 The usage of localized qualifiers is discouraged.

2206 DEPRECATED

2207 The qualifier type on which flavor (Translatable) is specified, is called the base qualifier of its localized
2208 qualifiers.

2209 The name of any localized qualifiers shall conform to the following formal syntax defined in ABNF:

```
2210 localized-qualifier-name = qualifier-name "_" locale
2211
2212 locale = language-code "_" country code
2213           ; the locale of the localized qualifier
```

2214 Where:

2215 `qualifier-name` is the name of the base qualifier of the localized qualifier

2216 `language-code` is a language code as defined in [ISO 639-1:2002](#), [ISO 639-2:1996](#), or [ISO 639-3:2007](#)
2217

2218 `country-code` is a country code as defined in [ISO 3166-1:2006](#), [ISO 3166-2:2007](#), or [ISO 3166-3:1999](#)
2219

2220 EXAMPLE:

2221 For the base qualifier named Description, the localized qualifier for Mexican Spanish language is named
2222 Description_es_MX.

2223 The string value of a localized qualifier shall be a translation of the string value of its base qualifier from
 2224 the language identified by the locale of the base qualifier into the language identified by the locale
 2225 specified in the name of the localized qualifier.

2226 For MOF, the locale of the base qualifier shall be the locale defined by the preceding #pragma locale
 2227 directive.

2228 For any localized qualifiers specified on a CIM element, a qualifier type with the same name (i.e.,
 2229 including the locale suffix) may be declared. If such a qualifier type is declared, its type, scope, flavor and
 2230 default value shall match the type, scope, flavor and default value of the base qualifier. If such a qualifier
 2231 type is not declared, it is implied from the qualifier type declaration of the base qualifier, with unchanged
 2232 type, scope, flavor and default value.

2233 **5.5.2 Meta Qualifiers**

2234 The following subclauses list the meta qualifiers required for all CIM-compliant implementations. Meta
 2235 qualifiers change the type of meta-element of the qualified schema element.

2236 **5.5.2.1 Association**

2237 The Association qualifier takes boolean values, has Scope (Association) and has Flavor
 2238 (DisableOverride). The default value is FALSE.

2239 This qualifier indicates that the class is defining an association, i.e., its type of meta-element becomes
 2240 Association.

2241 **5.5.2.2 Indication**

2242 The Indication qualifier takes boolean values, has Scope (Class, Indication) and has Flavor
 2243 (DisableOverride). The default value is FALSE.

2244 This qualifier indicates that the class is defining an indication, i.e., its type of meta-element becomes
 2245 Indication.

2246 **5.5.3 Standard Qualifiers**

2247 The following subclauses list the standard qualifiers required for all CIM-compliant implementations.
 2248 Additional qualifiers can be supplied by extension classes to provide instances of the class and other
 2249 operations on the class.

2250 Not all of these qualifiers can be used together. The following principles apply:

- 2251 • Not all qualifiers can be applied to all meta-model constructs. For each qualifier, the constructs
 2252 to which it applies are listed.
- 2253 • For a particular meta-model construct, such as associations, the use of the legal qualifiers may
 2254 be further constrained because some qualifiers are mutually exclusive or the use of one qualifier
 2255 implies restrictions on the value of another, and so on. These usage rules are documented in
 2256 the subclause for each qualifier.
- 2257 • Legal qualifiers are not inherited by meta-model constructs. For example, the MaxLen qualifier
 2258 that applies to properties is not inherited by references.
- 2259 • The meta-model constructs that can use a particular qualifier are identified for each qualifier.
 2260 For qualifiers such as Association (see 5.5.2), there is an implied usage rule that the meta
 2261 qualifier must also be present. For example, the implicit usage rule for the Aggregation qualifier
 2262 (see 5.5.3.3) is that the Association qualifier must also be present.

- 2263 • The allowed set of values for scope is (Class, Association, Indication, Property, Reference,
2264 Parameter, Method). Each qualifier has one or more of these scopes. If the scope is Class it
2265 does not apply to Association or Indication. If the scope is Property it does not apply to
2266 Reference.

2267 **5.5.3.1 Abstract**

2268 The Abstract qualifier takes boolean values, has Scope (Class, Association, Indication) and has Flavor
2269 (Restricted). The default value is FALSE.

2270 This qualifier indicates that the class is abstract and serves only as a base for new classes. It is not
2271 possible to create instances of such classes.

2272 **5.5.3.2 Aggregate**

2273 The Aggregate qualifier takes boolean values, has Scope (Reference) and has Flavor (DisableOverride).
2274 The default value is FALSE.

2275 The Aggregation and Aggregate qualifiers are used together. The Aggregation qualifier relates to the
2276 association, and the Aggregate qualifier specifies the parent reference.

2277 **5.5.3.3 Aggregation**

2278 The Aggregation qualifier takes boolean values, has Scope (Association) and has Flavor
2279 (DisableOverride). The default value is FALSE.

2280 The Aggregation qualifier indicates that the association is an aggregation.

2281 **5.5.3.4 ArrayType**

2282 The ArrayType qualifier takes string values, has Scope (Property, Parameter) and has Flavor
2283 (DisableOverride). The default value is "Bag".

2284 The ArrayType qualifier is the type of the qualified array. Valid values are "Bag", "Indexed," and
2285 "Ordered."

2286 For definitions of the array types, refer to 7.8.2.

2287 The ArrayType qualifier shall be applied only to properties and method parameters that are arrays
2288 (defined using the square bracket syntax specified in ANNEX A).

2289 The effective value of the ArrayType qualifier shall not change in the ancestry of the qualified element.
2290 This prevents incompatible changes in the behavior of the array element in subclasses.

2291 NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied
2292 default value to an explicitly specified value.

2293 **5.5.3.5 Bitmap**

2294 The Bitmap qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor
2295 (EnableOverride). The default value is NULL.

2296 The Bitmap qualifier indicates the bit positions that are significant in a bitmap. The bitmap is evaluated
2297 from the right, starting with the least significant value. This value is referenced as 0 (zero). For example,
2298 using a uint8 data type, the bits take the form Mxxx xxxL, where M and L designate the most and least
2299 significant bits, respectively. The least significant bits are referenced as 0 (zero), and the most significant
2300 bit is 7. The position of a specific value in the Bitmap array defines an index used to select a string literal
2301 from the BitValues array.

2302 The number of entries in the BitValues and Bitmap arrays shall match.

2303 **5.5.3.6 BitValues**

2304 The BitValues qualifier takes string array values, has Scope (Property, Parameter, Method) and has
2305 Flavor (EnableOverride, Translatable). The default value is NULL.

2306 The BitValues qualifier translates between a bit position value and an associated string. See 5.5.3.5 for
2307 the description for the Bitmap qualifier.

2308 The number of entries in the BitValues and Bitmap arrays shall match.

2309 **5.5.3.7 ClassConstraint**

2310 The ClassConstraint qualifier takes string array values, has Scope (Class, Association, Indication) and
2311 has Flavor (EnableOverride). The default value is NULL.

2312 The qualified element specifies one or more constraints that are defined in the OMG Object Constraint
2313 Language (OCL), as specified in the [Object Constraint Language](#) specification.

2314 The ClassConstraint array contains string values that specify OCL definition and invariant constraints.
2315 The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of the qualified
2316 class, association, or indication.

2317 OCL definition constraints define OCL attributes and OCL operations that are reusable by other OCL
2318 constraints in the same OCL context.

2319 The attributes and operations in the OCL definition constraints shall be visible for:

- 2320 • OCL definition and invariant constraints defined in subsequent entries in the same
2321 ClassConstraint array
- 2322 • OCL constraints defined in PropertyConstraint qualifiers on properties and references in a class
2323 whose value (specified or inherited) of the ClassConstraint qualifier defines the OCL definition
2324 constraint
- 2325 • Constraints defined in MethodConstraint qualifiers on methods defined in a class whose value
2326 (specified or inherited) of the ClassConstraint qualifier defines the OCL definition constraint

2327 A string value specifying an OCL definition constraint shall conform to the following formal syntax defined
2328 in ABNF (whitespace allowed):

```
2329 ocl_definition_string = "def" [ocl_name] ":" ocl_statement
```

2330 Where:

2331 `ocl_name` is the name of the OCL constraint.

2332 `ocl_statement` is the OCL statement of the definition constraint, which defines the reusable
2333 attribute or operation.

2334 An OCL invariant constraint is expressed as a typed OCL expression that specifies whether the constraint
2335 is satisfied. The type of the expression shall be boolean. The invariant constraint shall be satisfied at any
2336 time in the lifetime of the instance.

2337 A string value specifying an OCL invariant constraint shall conform to the following formal syntax defined
2338 in ABNF (whitespace allowed):

```
2339 ocl_invariant_string = "inv" [ocl_name] ":" ocl_statement
```

2340 Where:

2341 ocl_name is the name of the OCL constraint.

2342 ocl_statement is the OCL statement of the invariant constraint, which defines the boolean
2343 expression.

2344 EXAMPLE 1: For example, to check that both property x and property y cannot be NULL in any instance of a class,
2345 use the following qualifier, defined on the class:

```
2346 ClassConstraint {
2347     "inv: not (self.x.ocIsUndefined() and self.y.ocIsUndefined())"
2348 }
```

2349 EXAMPLE 2: The same check can be performed by first defining OCL attributes. Also, the invariant constraint is
2350 named in the following example:

```
2351 ClassConstraint {
2352     "def: xNull : Boolean = self.x.ocIsUndefined()",
2353     "def: yNull : Boolean = self.y.ocIsUndefined()",
2354     "inv xyNullCheck: xNull = false or yNull = false)"
2355 }
```

2356 **5.5.3.8 Composition**

2357 The Composition qualifier takes boolean values, has Scope (Association) and has Flavor
2358 (DisableOverride). The default value is FALSE.

2359 The Composition qualifier refines the definition of an aggregation association, adding the semantics of a
2360 whole-part/compositional relationship to distinguish it from a collection or basic aggregation. This
2361 refinement is necessary to map CIM associations more precisely into UML where whole-part relationships
2362 are considered compositions. The semantics conveyed by composition align with that of the [Unified](#)
2363 [Modeling Language: Superstructure](#). Following is a quote (with emphasis added) from its section 7.3.3:

2364 "Composite aggregation is a strong form of aggregation that requires a part instance be included
2365 in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted
2366 with it."

2367 Use of this qualifier imposes restrictions on the membership of the 'collecting' object (the whole). Care
2368 should be taken when entities are added to the aggregation, because they shall be "parts" of the whole.
2369 Also, if the collecting entity (the whole) is deleted, it is the responsibility of the implementation to dispose
2370 of the parts. The behavior may vary with the type of collecting entity whether the parts are also deleted.
2371 This is very different from that of a collection, because a collection may be removed without deleting the
2372 entities that are collected.

2373 The Aggregation and Composition qualifiers are used together. Aggregation indicates the general nature
2374 of the association, and Composition indicates more specific semantics of whole-part relationships. This
2375 duplication of information is necessary because Composition is a more recent addition to the list of
2376 qualifiers. Applications can be built only on the basis of the earlier Aggregation qualifier.

2377 **5.5.3.9 Correlatable**

2378 The Correlatable qualifier takes string array values, has Scope (Property) and has Flavor
2379 (EnableOverride). The default value is NULL.

2380 The Correlatable qualifier is used to define sets of properties that can be compared to determine if two
2381 CIM instances represent the same resource entity. For example, these instances may cross
2382 logical/physical boundaries, CIM server scopes, or implementation interfaces.

2383 The sets of properties to be compared are defined by first specifying the organization in whose context
2384 the set exists (organization_name), and then a set name (set_name). In addition, a property is given a

2385 role name (role_name) to allow comparisons across the CIM Schema (that is, where property names may
2386 vary although the semantics are consistent).

2387 The value of each entry in the Correlatable qualifier string array shall follow the formal syntax defined in
2388 ABNF:

```
2389 correlatablePropertyID = organization_name ":" set_name ":" role_name
```

2390 The determination whether two CIM instances represent the same resource entity is done by comparing
2391 one or more property values of each instance (where the properties are tagged by their role name), as
2392 follows: The property values of all role names within at least one matching organization name / set name
2393 pair shall match in order to conclude that the two instances represent the same resource entity.
2394 Otherwise, no conclusion can be reached and the instances may or may not represent the same resource
2395 entity.

2396 correlatablePropertyID values shall be compared case-insensitively. For example,

```
2397 "Acme:Set1:Role1" and "ACME:set1:role1"
```

2398 are considered matching.

2399 NOTE: The values of any string properties in CIM are defined to be compared case-sensitively.

2400 To assure uniqueness of a correlatablePropertyID:

- 2401 • organization_name shall include a copyrighted, trademarked or otherwise unique name that is
2402 owned by the business entity defining set_name, or is a registered ID that is assigned to the
2403 business entity by a recognized global authority. organization_name shall not contain a colon
2404 (":"). For DMTF defined correlatablePropertyID values, the organization_name shall be
2405 "CIM".
- 2406 • set_name shall be unique within the context of organization_name and identifies a specific set
2407 of correlatable properties. set_name shall not contain a colon (":").
- 2408 • role_name shall be unique within the context of organization_name and set_name and identifies
2409 the semantics or role that the property plays within the Correlatable comparison.

2410 The Correlatable qualifier may be defined on only a single class. In this case, instances of only that class
2411 are compared. However, if the same correlation set (defined by organization_name and set_name) is
2412 specified on multiple classes, then comparisons can be done across those classes.

2413 EXAMPLE: As an example, assume that instances of two classes can be compared: Class1 with properties PropA,
2414 PropB, and PropC, and Class2 with properties PropX, PropY and PropZ. There are two correlation sets
2415 defined, one set with two properties that have the role names Role1 and Role2, and the other set with
2416 one property with the role name OnlyRole. The following MOF represents this example:

```
2417 Class1 {
2418
2419     [Correlatable {"Acme:Set1:Role1"}]
2420     string PropA;
2421
2422     [Correlatable {"Acme:Set2:OnlyRole"}]
2423     string PropB;
2424
2425     [Correlatable {"Acme:Set1:Role2"}]
2426     string PropC;
2427 };
2428
2429 Class2 {
```

```
2430
2431     [Correlatable {"Acme:Set1:Role1"}]
2432     string PropX;
2433
2434     [Correlatable {"Acme:Set2:OnlyRole"}]
2435     string PropY;
2436
2437     [Correlatable {"Acme:Set1:Role2"}]
2438     string PropZ;
2439 };
```

2440 Following the comparison rules defined above, one can conclude that an instance of Class1 and an
2441 instance of Class2 represent the same resource entity if PropB and PropY's values match, or if
2442 PropA/PropX and PropC/PropZ's values match, respectively.

2443 The Correlatable qualifier can be used to determine if multiple CIM instances represent the same
2444 underlying resource entity. Some may wonder if an instance's key value (such as InstanceID) is meant to
2445 perform the same role. This is not the case. InstanceID is merely an opaque identifier of a CIM instance,
2446 whereas Correlatable is not opaque and can be used to draw conclusions about the identity of the
2447 underlying resource entity of two or more instances.

2448 DMTF-defined Correlatable qualifiers are defined in the CIM Schema on a case-by-case basis. There is
2449 no central document that defines them.

2450 **5.5.3.10 Counter**

2451 The Counter qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor
2452 (EnableOverride). The default value is FALSE.

2453 The Counter qualifier applies only to unsigned integer types.

2454 It represents a non-negative integer that monotonically increases until it reaches a maximum value of
2455 $2^n - 1$, when it wraps around and starts increasing again from zero. N can be 8, 16, 32, or 64 depending
2456 on the data type of the object to which the qualifier is applied. Counters have no defined initial value, so a
2457 single value of a counter generally has no information content.

2458 **5.5.3.11 Deprecated**

2459 The Deprecated qualifier takes string array values, has Scope (Class, Association, Indication, Property,
2460 Reference, Parameter, Method) and has Flavor (Restricted). The default value is NULL.

2461 The Deprecated qualifier indicates that the CIM element (for example, a class or property) that the
2462 qualifier is applied to is considered deprecated. The qualifier may specify replacement elements. Existing
2463 CIM servers shall continue to support the deprecated element so that current CIM clients do not break.
2464 Existing CIM servers should add support for any replacement elements. A deprecated element should not
2465 be used in new CIM clients. Existing and new CIM clients shall tolerate the deprecated element and
2466 should move to any replacement elements as soon as possible. The deprecated element may be
2467 removed in a future major version release of the CIM schema, such as CIM 2.x to CIM 3.0.

2468 The qualifier acts inclusively. Therefore, if a class is deprecated, all the properties, references, and
2469 methods in that class are also considered deprecated. However, no subclasses or associations or
2470 methods that reference that class are deprecated unless they are explicitly qualified as such. For clarity
2471 and to specify replacement elements, all such implicitly deprecated elements should be specifically
2472 qualified as deprecated.

2473 The Deprecated qualifier's string value should specify one or more replacement elements. Replacement
2474 elements shall be specified using the following formal syntax defined in ABNF:

2475 `deprecatedEntry = className [[embeddedInstancePath] "." elementSpec]`

2476 where:

2477 `elementSpec = propertyName / methodName "(" [parameterName *("," parameterName)] ")"`

2478 is a specification of the replacement element.

2479 `embeddedInstancePath = 1*("." propertyName)`

2480 is a specification of a path through embedded instances.

2481 The qualifier is defined as a string array so that a single element can be replaced by multiple elements.

2482 If there is no replacement element, then the qualifier string array shall contain a single entry with the
2483 string "No value".

2484 When an element is deprecated, its description shall indicate why it is deprecated and how any
2485 replacement elements are used. Following is an acceptable example description:

2486 "The X property is deprecated in lieu of the Y method defined in this class because the property actually
2487 causes a change of state and requires an input parameter."

2488 The parameters of the replacement method may be omitted.

2489 NOTE 1: Replacing a deprecated element with a new element results in duplicate representations of the element.
2490 This is of particular concern when deprecated classes are replaced by new classes and instances may be duplicated.
2491 To allow a CIM client to detect such duplication, implementations should document (in a ReadMe, MOF, or other
2492 documentation) how such duplicate instances are detected.

2493 NOTE 2: Key properties may be deprecated, but they shall continue to be key properties and shall satisfy all rules for
2494 key properties. When a key property is no longer intended to be a key, only one option is available. It is necessary to
2495 deprecate the entire class and therefore its properties, methods, references, and so on, and to define a new class
2496 with the changed key structure.

2497 **5.5.3.12 Description**

2498 The Description qualifier takes string values, has Scope (Class, Association, Indication, Property,
2499 Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is
2500 NULL.

2501 The Description qualifier describes a named element.

2502 **5.5.3.13 DisplayName**

2503 The DisplayName qualifier takes string values, has Scope (Class, Association, Indication, Property,
2504 Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is
2505 NULL.

2506 The DisplayName qualifier defines a name that is displayed on a user interface instead of the actual
2507 name of the element.

2508 **5.5.3.14 DN**

2509 The DN qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor
2510 (DisableOverride). The default value is FALSE.

2511 When applied to a string element, the DN qualifier specifies that the string shall be a distinguished name
2512 as defined in Section 9 of [ITU X.501](#) and the string representation defined in [RFC2253](#). This qualifier shall
2513 not be applied to qualifiers that are not of the intrinsic data type string.

2514 5.5.3.15 EmbeddedInstance

2515 The EmbeddedInstance qualifier takes string values, has Scope (Property, Parameter, Method) and has
2516 Flavor (EnableOverride). The default value is NULL.

2517 A non-NULL effective value of this qualifier indicates that the qualified string typed element contains an
2518 embedded instance. The encoding of the instance contained in the string typed element qualified by
2519 EmbeddedInstance shall follow the rules defined in ANNEX F.

2520 This qualifier may be used only on elements of string type.

2521 If not NULL the qualifier value shall specify the name of a CIM class in the same namespace as the class
2522 owning the qualified element. The embedded instance shall be an instance of the specified class,
2523 including instances of its subclasses.

2524 The value of the EmbeddedInstance qualifier may be changed in subclasses to narrow the originally
2525 specified class to one of its subclasses. Other than that, the effective value of the EmbeddedInstance
2526 qualifier shall not change in the ancestry of the qualified element. This prevents incompatible changes
2527 between representing and not representing an embedded instance in subclasses.

2528 See ANNEX F for examples.

2529 5.5.3.16 EmbeddedObject

2530 The EmbeddedObject qualifier takes boolean values, has Scope (Property, Parameter, Method) and has
2531 Flavor (DisableOverride). The default value is FALSE.

2532 This qualifier indicates that the qualified string typed element contains an encoding of an instance's data
2533 or an encoding of a class definition. The encoding of the object contained in the string typed element
2534 qualified by EmbeddedObject shall follow the rules defined in ANNEX F.

2535 This qualifier may be used only on elements of string type.

2536 The effective value of the EmbeddedObject qualifier shall not change in the ancestry of the qualified
2537 element. This prevents incompatible changes between representing and not representing an embedded
2538 object in subclasses.

2539 NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied
2540 default value to an explicitly specified value.

2541 See ANNEX F for examples.

2542 5.5.3.17 Exception

2543 The Exception qualifier takes boolean values, has Scope (Class, Indication) and has Flavor
2544 (DisableOverride). The default value is FALSE.

2545 This qualifier indicates that the class and all subclasses of this class describe transient exception
2546 information. The definition of this qualifier is identical to that of the Abstract qualifier except that it cannot
2547 be overridden. It is not possible to create instances of exception classes.

2548 The Exception qualifier denotes a class hierarchy that defines transient (very short-lived) exception
2549 objects. Instances of Exception classes communicate exception information between CIM entities. The
2550 Exception qualifier cannot be used with the Abstract qualifier. The subclass of an exception class shall be
2551 an exception class.

2552 5.5.3.18 Experimental

2553 The Experimental qualifier takes boolean values, has Scope (Class, Association, Indication, Property,
2554 Reference, Parameter, Method) and has Flavor (Restricted). The default value is FALSE.

2555 If the Experimental qualifier is specified, the qualified element has experimental status. The implications
2556 of experimental status are specified by the schema owner.

2557 In a DMTF-produced schema, experimental elements are subject to change and are not part of the final
2558 schema. In particular, the requirement to maintain backwards compatibility across minor schema versions
2559 does not apply to experimental elements. Experimental elements are published for developing
2560 implementation experience. Based on implementation experience, changes may occur to this element in
2561 future releases, it may be standardized "as is," or it may be removed. An implementation does not have to
2562 support an experimental feature to be compliant to a DMTF-published schema.

2563 When applied to a class, the Experimental qualifier conveys experimental status to the class itself, as well
2564 as to all properties and features defined on that class. Therefore, if a class already bears the
2565 Experimental qualifier, it is unnecessary also to apply the Experimental qualifier to any of its properties or
2566 features, and such redundant use is discouraged.

2567 No element shall be both experimental and deprecated (as with the Deprecated qualifier). Experimental
2568 elements whose use is considered undesirable should simply be removed from the schema.

2569 5.5.3.19 Gauge

2570 The Gauge qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor
2571 (EnableOverride). The default value is FALSE.

2572 The Gauge qualifier is applicable only to unsigned integer types. It represents an integer that may
2573 increase or decrease in any order of magnitude.

2574 The value of a gauge is capped at the implied limits of the property's data type. If the information being
2575 modeled exceeds an implied limit, the value represented is that limit. Values do not wrap. For unsigned
2576 integers, the limits are zero (0) to 2^n-1 , inclusive. For signed integers, the limits are $-(2^{n-1})$ to
2577 $2^{n-1}-1$, inclusive. N can be 8, 16, 32, or 64 depending on the data type of the property to which the
2578 qualifier is applied.

2579 5.5.3.20 In

2580 The In qualifier takes boolean values, has Scope (Parameter) and has Flavor (DisableOverride). The
2581 default value is TRUE.

2582 This qualifier indicates that the qualified parameter is used to pass values to a method.

2583 The effective value of the In qualifier shall not change in the ancestry of the qualified parameter. This
2584 prevents incompatible changes in the direction of parameters in subclasses.

2585 NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied
2586 default value to an explicitly specified value.

2587 5.5.3.21 IsPUnit

2588 The IsPUnit qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor
2589 (EnableOverride). The default value is FALSE.

2590 The qualified string typed property, method return value, or method parameter represents a programmatic
2591 unit of measure. The value of the string element follows the syntax for programmatic units.

2592 The qualifier must be used on string data types only. A value of NULL for the string element indicates that
2593 the programmatic unit is unknown. The syntax for programmatic units is defined in ANNEX C.

2594 **5.5.3.22 Key**

2595 The Key qualifier takes boolean values, has Scope (Property, Reference) and has Flavor
2596 (DisableOverride). The default value is FALSE.

2597 The property or reference is part of the model path (see 8.2.5 for information on the model path). If more
2598 than one property or reference has the Key qualifier, then all such elements collectively form the key (a
2599 compound key).

2600 The values of key properties and key references are determined once at instance creation time and shall
2601 not be modified afterwards. Properties of an array type shall not be qualified with Key. Properties qualified
2602 with EmbeddedObject or EmbeddedInstance shall not be qualified with Key. Key properties and Key
2603 references shall not be NULL.

2604 **5.5.3.23 MappingStrings**

2605 The MappingStrings qualifier takes string array values, has Scope (Class, Association, Indication,
2606 Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is NULL.

2607 This qualifier indicates mapping strings for one or more management data providers or agents. See 5.5.6
2608 for details.

2609 **5.5.3.24 Max**

2610 The Max qualifier takes uint32 values, has Scope (Reference) and has Flavor (EnableOverride). The
2611 default value is NULL.

2612 The Max qualifier specifies the maximum cardinality of the reference, which is the maximum number of
2613 values a given reference may have for each set of other reference values in the association. For example,
2614 if an association relates A instances to B instances, and there shall be at most one A instance for each B
2615 instance, then the reference to A should have a Max(1) qualifier.

2616 The NULL value means that the maximum cardinality is unlimited.

2617 **5.5.3.25 MaxLen**

2618 The MaxLen qualifier takes uint32 values, has Scope (Property, Parameter, Method) and has Flavor
2619 (EnableOverride). The default value is NULL.

2620 The MaxLen qualifier specifies the maximum length, in characters, of a string data item. MaxLen may be
2621 used only on string data types. If MaxLen is applied to CIM elements with a string array data type, it
2622 applies to every element of the array. A value of NULL implies unlimited length.

2623 An overriding property that specifies the MAXLEN qualifier must specify a maximum length no greater
2624 than the maximum length for the property being overridden.

2625 **5.5.3.26 MaxValue**

2626 The MaxValue qualifier takes sint64 values, has Scope (Property, Parameter, Method) and has Flavor
2627 (EnableOverride). The default value is NULL.

2628 The MaxValue qualifier specifies the maximum value of this element. MaxValue may be used only on
2629 numeric data types. If MaxValue is applied to CIM elements with a numeric array data type, it applies to
2630 every element of the array. A value of NULL means that the maximum value is the highest value for the
2631 data type.

2632 An overriding property that specifies the MaxValue qualifier must specify a maximum value no greater
2633 than the maximum value of the property being overridden.

2634 **5.5.3.27 MethodConstraint**

2635 The MethodConstraint qualifier takes string array values, has Scope (Method) and has Flavor
2636 (EnableOverride). The default value is NULL.

2637 The qualified element specifies one or more constraints, which are defined using the OMG Object
2638 Constraint Language (OCL), as specified in the [Object Constraint Language](#) specification.

2639 The MethodConstraint array contains string values that specify OCL precondition, postcondition, and
2640 body constraints.

2641 The OCL context of these constraints (that is, what "self" in OCL refers to) is the object on which the
2642 qualified method is invoked.

2643 An OCL precondition constraint is expressed as a typed OCL expression that specifies whether the
2644 precondition is satisfied. The type of the expression shall be boolean. For the method to complete
2645 successfully, all preconditions of a method shall be satisfied before it is invoked.

2646 A string value specifying an OCL precondition constraint shall conform to the formal syntax defined in
2647 ABNF (whitespace allowed):

```
2648 ocl_precondition_string = "pre" [ocl_name] ":" ocl_statement
```

2649 Where:

2650 `ocl_name` is the name of the OCL constraint.

2651 `ocl_statement` is the OCL statement of the precondition constraint, which defines the boolean
2652 expression.

2653 An OCL postcondition constraint is expressed as a typed OCL expression that specifies whether the
2654 postcondition is satisfied. The type of the expression shall be boolean. All postconditions of the method
2655 shall be satisfied immediately after successful completion of the method.

2656 A string value specifying an OCL post-condition constraint shall conform to the following formal syntax
2657 defined in ABNF (whitespace allowed):

```
2658 ocl_postcondition_string = "post" [ocl_name] ":" ocl_statement
```

2659 Where:

2660 `ocl_name` is the name of the OCL constraint.

2661 `ocl_statement` is the OCL statement of the post-condition constraint, which defines the boolean
2662 expression.

2663 An OCL body constraint is expressed as a typed OCL expression that specifies the return value of a
2664 method. The type of the expression shall conform to the CIM data type of the return value. Upon
2665 successful completion, the return value of the method shall conform to the OCL expression.

2666 A string value specifying an OCL body constraint shall conform to the following formal syntax defined in
2667 ABNF (whitespace allowed):

```
2668 ocl_body_string = "body" [ocl_name] ":" ocl_statement
```

2669 Where:

2670 `ocl_name` is the name of the OCL constraint.

2671 `ocl_statement` is the OCL statement of the body constraint, which defines the method return
2672 value.

2673 EXAMPLE: The following qualifier defined on the `RequestedStateChange()` method of the
2674 `CIM_EnabledLogicalElement` class specifies that if a `Job` parameter is returned as not NULL, then an
2675 `CIM_OwningJobElement` association must exist between the `CIM_EnabledLogicalElement` class and the `Job`.

```
2676 MethodConstraint {
2677     "post AssociatedJob: "
2678         "not Job.oclIsUndefined() "
2679         "implies "
2680         "self.cIM_OwningJobElement.OwnedElement = Job"
2681 }
```

2682 5.5.3.28 Min

2683 The Min qualifier takes uint32 values, has Scope (Reference) and has Flavor (EnableOverride). The
2684 default value is 0.

2685 The Min qualifier specifies the minimum cardinality of the reference, which is the minimum number of
2686 values a given reference may have for each set of other reference values in the association. For example,
2687 if an association relates A instances to B instances and there shall be at least one A instance for each B
2688 instance, then the reference to A should have a `Min(1)` qualifier.

2689 The qualifier value shall not be NULL.

2690 5.5.3.29 MinLen

2691 The MinLen qualifier takes uint32 values, has Scope (Property, Parameter, Method) and has Flavor
2692 (EnableOverride). The default value is 0.

2693 The MinLen qualifier specifies the minimum length, in characters, of a string data item. MinLen may be
2694 used only on string data types. If MinLen is applied to CIM elements with a string array data type, it
2695 applies to every element of the array. The NULL value is not allowed for MinLen.

2696 An overriding property that specifies the MinLen qualifier must specify a minimum length no smaller than
2697 the minimum length of the property being overridden.

2698 5.5.3.30 MinValue

2699 The MinValue qualifier takes sint64 values, has Scope (Property, Parameter, Method) and has Flavor
2700 (EnableOverride). The default value is NULL.

2701 The MinValue qualifier specifies the minimum value of this element. MinValue may be used only on
2702 numeric data types. If MinValue is applied to CIM elements with a numeric array data type, it applies to
2703 every element of the array. A value of NULL means that the minimum value is the lowest value for the
2704 data type.

2705 An overriding property that specifies the MinValue qualifier must specify a minimum value no smaller than
2706 the minimum value of the property being overridden.

2707 5.5.3.31 ModelCorrespondence

2708 The ModelCorrespondence qualifier takes string array values, has Scope (Class, Association, Indication,
2709 Property, Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is NULL.

2710 The ModelCorrespondence qualifier indicates a correspondence between two elements in the CIM
 2711 schema. The referenced elements shall be defined in a standard or extension MOF file, such that the
 2712 correspondence can be examined. If possible, forward referencing of elements should be avoided.

2713 Object elements are identified using the following formal syntax defined in ABNF:

```
2714 modelCorrespondenceEntry = className [ *( "." ( propertyName / referenceName ) )
2715                               [ "." methodName
2716                               [ "(" [ parameterName *( "," parameterName ) ] ")" ] ] ]
```

2717 The basic relationship between the referenced elements is a "loose" correspondence, which simply
 2718 indicates that the elements are coupled. This coupling may be unidirectional. Additional qualifiers may be
 2719 used to describe a tighter coupling.

2720 The following list provides examples of several correspondences found in CIM and vendor schemas:

- 2721 • A vendor defines an Indication class corresponding to a particular CIM property or method so
 2722 that Indications are generated based on the values or operation of the property or method. In
 2723 this case, the ModelCorrespondence may only be on the vendor's Indication class, which is an
 2724 extension to CIM.
- 2725 • A property provides more information for another. For example, an enumeration has an allowed
 2726 value of "Other", and another property further clarifies the intended meaning of "Other." In
 2727 another case, a property specifies status and another property provides human-readable strings
 2728 (using an array construct) expanding on this status. In these cases, ModelCorrespondence is
 2729 found on both properties, each referencing the other. Also, referenced array properties may not
 2730 be ordered but carry the default ArrayType qualifier definition of "Bag."
- 2731 • A property is defined in a subclass to supplement the meaning of an inherited property. In this
 2732 case, the ModelCorrespondence is found only on the construct in the subclass.
- 2733 • Multiple properties taken together are needed for complete semantics. For example, one
 2734 property may define units, another property may define a multiplier, and another property may
 2735 define a specific value. In this case, ModelCorrespondence is found on all related properties,
 2736 each referencing all the others.
- 2737 • Multi-dimensional arrays are desired. For example, one array may define names while another
 2738 defines the name formats. In this case, the arrays are each defined with the
 2739 ModelCorrespondence qualifier, referencing the other array properties or parameters. Also, they
 2740 are indexed and they carry the ArrayType qualifier with the value "Indexed."

2741 The semantics of the correspondence are based on the elements themselves. ModelCorrespondence is
 2742 only a hint or indicator of a relationship between the elements.

2743 **5.5.3.32 NonLocal (removed)**

2744 This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0
 2745 of this document.

2746 **5.5.3.33 NonLocalType (removed)**

2747 This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0
 2748 of this document.

2749 **5.5.3.34 NullValue**

2750 The NullValue qualifier takes string values, has Scope (Property) and has Flavor (DisableOverride). The
 2751 default value is NULL.

2752 The NullValue qualifier defines a value that indicates that the associated property is NULL. That is, the
2753 property is considered to have a valid or meaningful value.

2754 The NullValue qualifier may be used only with properties that have string and integer values. When used
2755 with an integer type, the qualifier value is a MOF decimal value as defined by the `decimalValue` ABNF
2756 rule defined in ANNEX A.

2757 The content, maximum number of digits, and represented value are constrained by the data type of the
2758 qualified property.

2759 This qualifier cannot be overridden because it seems unreasonable to permit a subclass to return a
2760 different null value than that of the superclass.

2761 **5.5.3.35 OctetString**

2762 The OctetString qualifier takes boolean values, has Scope (Property, Parameter, Method) and has Flavor
2763 (DisableOverride). The default value is FALSE.

2764 This qualifier indicates that the qualified element is an octet string. An octet string is a sequence of octets
2765 and allows the representation of binary data.

2766 The OctetString qualifier shall be specified only on elements of type array of uint8 or array of string.

2767 When specified on elements of type array of uint8, the OctetString qualifier indicates that the entire array
2768 represents a single octet string. The first four array entries shall represent a length field, and any
2769 subsequent entries shall represent the octets in the octet string. The four uint8 values in the length field
2770 shall be interpreted as a 32-bit unsigned number where the first array entry is the most significant byte.
2771 The number represented by the length field shall be the number of octets in the octet string plus four. For
2772 example, the empty octet string is represented as { 0x00, 0x00, 0x00, 0x04 }.

2773 When specified on elements of type array of string, the OctetString qualifier indicates that each array
2774 entry represents a separate octet string. The string value of each array entry shall be interpreted as a
2775 textual representation of the octet string. The string value of each array entry shall conform to the
2776 following formal syntax defined in ABNF:

```
2777 "0x" 4*( hexDigit hexDigit )
```

2778 The first four pairs of hexadecimal digits of the string value shall represent a length field, and any
2779 subsequent pairs shall represent the octets in the octet string. The four pairs of hexadecimal digits in the
2780 length field shall be interpreted as a 32-bit unsigned number where the first pair is the most significant
2781 byte. The number represented by the length field shall be the number of octets in the octet string plus
2782 four. For example, the empty octet string is represented as "0x00000004".

2783 The effective value of the OctetString qualifier shall not change in the ancestry of the qualified element.
2784 This prevents incompatible changes in the interpretation of the qualified element in subclasses.

2785 NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied
2786 default value to an explicitly specified value.

2787 **5.5.3.36 Out**

2788 The Out qualifier takes boolean values, has Scope (Parameter) and has Flavor (DisableOverride). The
2789 default value is FALSE.

2790 This qualifier indicates that the qualified parameter is used to return values from a method.

2791 The effective value of the Out qualifier shall not change in the ancestry of the qualified parameter. This
2792 prevents incompatible changes in the direction of parameters in subclasses.

2793 NOTE: The DisableOverride flavor alone is not sufficient to ensure this, since it allows one change from the implied
2794 default value to an explicitly specified value.

2795 **5.5.3.37 Override**

2796 The Override qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor
2797 (Restricted). The default value is NULL.

2798 If non-NULL, the qualified element in the derived (containing) class takes the place of another element (of
2799 the same name) defined in the ancestry of that class.

2800 The flavor of the qualifier is defined as 'Restricted' so that the Override qualifier is not repeated in
2801 (inherited by) each subclass. The effect of the override is inherited, but not the identification of the
2802 Override qualifier itself. This enables new Override qualifiers in subclasses to be easily located and
2803 applied.

2804 An effective value of NULL (the default) indicates that the element is not overriding any element. If not
2805 NULL, the value shall conform to the following formal syntax defined in ABNF:

```
2806 [ className "." ] IDENTIFIER
```

2807 where IDENTIFIER shall be the name of the overridden element and if present, className shall
2808 be the name of a class in the ancestry of the derived class. The className ABNF rule shall be
2809 present if the class exposes more than one element with the same name (see 7.5.1).

2810 If className is omitted, the overridden element is found by searching the ancestry of the class until a
2811 definition of an appropriately-named subordinate element (of the same meta-schema class) is found.

2812 If className is specified, the element being overridden is found by searching the named class and its
2813 ancestry until a definition of an element of the same name (of the same meta-schema class) is found.

2814 The Override qualifier may only refer to elements of the same meta-schema class. For example,
2815 properties can only override properties, etc. An element's name or signature shall not be changed when
2816 overriding.

2817 **5.5.3.38 Propagated**

2818 The Propagated qualifier takes string values, has Scope (Property) and has Flavor (DisableOverride).
2819 The default value is NULL.

2820 When the Propagated qualifier is specified with a non-NULL value on a property, the Key qualifier shall be
2821 specified with a value of TRUE on the qualified property.

2822 A non-NULL value of the Propagated qualifier indicates that the value of the qualified key property is
2823 propagated from a property in another instance that is associated via a weak association. That associated
2824 instance is referred to as the scoping instance of the instance receiving the property value.

2825 A non-NULL value of the Propagated qualifier shall identify the property in the scoping instance and shall
2826 conform to the formal syntax defined in ABNF:

```
2827 [ className "." ] propertyName
```

2828 where propertyName is the name of the property in the scoping instance, and className is the name
2829 of a class exposing that property. The specification of a class name may be needed in order to
2830 disambiguate like-named properties in associations with an arity of three or higher. It is recommended to
2831 specify the class name in any case.

2832 For a description of the concepts of weak associations and key propagation as well as further rules
2833 around them, see 8.2

2834 **5.5.3.39 PropertyConstraint**

2835 The PropertyConstraint qualifier takes string array values, has Scope (Property, Reference) and has
2836 Flavor (EnableOverride). The default value is NULL.

2837 The qualified element specifies one or more constraints that are defined using the Object Constraint
2838 Language (OCL) as specified in the [Object Constraint Language](#) specification.

2839 The PropertyConstraint array contains string values that specify OCL initialization and derivation
2840 constraints. The OCL context of these constraints (that is, what "self" in OCL refers to) is an instance of
2841 the class, association, or indication that exposes the qualified property or reference.

2842 An OCL initialization constraint is expressed as a typed OCL expression that specifies the permissible
2843 initial value for a property. The type of the expression shall conform to the CIM data type of the property.

2844 A string value specifying an OCL initialization constraint shall conform to the following formal syntax
2845 defined in ABNF (whitespace allowed):

```
2846 ocl_initialization_string = "init" ":" ocl_statement
```

2847 Where:

2848 `ocl_statement` is the OCL statement of the initialization constraint, which defines the typed
2849 expression.

2850 An OCL derivation constraint is expressed as a typed OCL expression that specifies the permissible
2851 value for a property at any time in the lifetime of the instance. The type of the expression shall conform to
2852 the CIM data type of the property.

2853 A string value specifying an OCL derivation constraint shall conform to the following formal syntax defined
2854 in ABNF (whitespace allowed):

```
2855 ocl_derivation_string = "derive" ":" ocl_statement
```

2856 Where:

2857 `ocl_statement` is the OCL statement of the derivation constraint, which defines the typed
2858 expression.

2859 For example, PolicyAction has a SystemName property that must be set to the name of the system
2860 associated with CIM_PolicySetInSystem. The following qualifier defined on
2861 CIM_PolicyAction.SystemName specifies that constraint:

```
2862 PropertyConstraint {  
2863     "derive: self.CIM_PolicySetInSystem.Antecedent.Name"  
2864 }
```

2865 A default value defined on a property also represents an initialization constraint, and no more than one
2866 initialization constraint is allowed on a property, as defined in 5.1.2.8.

2867 No more than one derivation constraint is allowed on a property, as defined in 5.1.2.8.

2868 **5.5.3.40 PUnit**

2869 The PUnit qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor
2870 (EnableOverride). The default value is NULL.

2871 The PUnit qualifier indicates the programmatic unit of measure of the schema element. The qualifier
2872 value shall follow the syntax for programmatic units, as defined in ANNEX C.

2873 The PUnit qualifier shall be specified only on schema elements of a numeric datatype. An effective value
2874 of NULL indicates that a programmatic unit is unknown for or not applicable to the schema element.

2875 String typed schema elements that are used to represent numeric values in a string format cannot have
2876 the PUnit qualifier specified, since the reason for using string typed elements to represent numeric values
2877 is typically that the type of value changes over time, and hence a programmatic unit for the element
2878 needs to be able to change along with the type of value. This can be achieved with a companion schema
2879 element whose value specifies the programmatic unit in case the first schema element holds a numeric
2880 value. This companion schema element would be string typed and the IsPUnit qualifier be set to true.

2881 **5.5.3.41 Read**

2882 The Read qualifier takes boolean values, has Scope (Property) and has Flavor (EnableOverride). The
2883 default value is TRUE.

2884 The Read qualifier indicates that the property is readable.

2885 **5.5.3.42 Required**

2886 The Required qualifier takes boolean values, has Scope (Property, Reference, Parameter, Method) and
2887 has Flavor (DisableOverride). The default value is FALSE.

2888 A non-NULL value is required for the element. For CIM elements with an array type, the Required
2889 qualifier affects the array itself, and the elements of the array may be NULL regardless of the Required
2890 qualifier.

2891 Properties of a class that are inherent characteristics of a class and identify that class are such properties
2892 as domain name, file name, burned-in device identifier, IP address, and so on. These properties are likely
2893 to be useful for CIM clients as query entry points that are not KEY properties but should be Required
2894 properties.

2895 References of an association that are not KEY references shall be Required references. There are no
2896 particular usage rules for using the Required qualifier on parameters of a method outside of the meaning
2897 defined in this clause.

2898 A property that overrides a required property shall not specify REQUIRED(false).

2899 Compatible schema changes may add the Required qualifier to method output parameters, methods (i.e.,
2900 their return values) and properties that may only be read. Compatible schema changes may remove the
2901 Required qualifier from method input parameters and properties that may only be written. If such
2902 compatible schema changes are done, the description of the changed schema element should indicate
2903 the schema version in which the change was made. This information can be used for example by
2904 management profile implementations in order to decide whether it is appropriate to implement a schema
2905 version higher than the one minimally required by the profile, and by CIM clients in order to decide
2906 whether they need to support both behaviors.

2907 **5.5.3.43 Revision (deprecated)**

2908 **DEPRECATED**

2909 The Revision qualifier is deprecated (See 5.5.3.53 for the description of the Version qualifier).

2910 The Revision qualifier takes string values, has Scope (Class, Association, Indication) and has Flavor
2911 (EnableOverride, Translatable). The default value is NULL.

2912 The Revision qualifier provides the minor revision number of the schema object.

2913 The Version qualifier shall be present to supply the major version number when the Revision qualifier is
2914 used.

2915 **DEPRECATED**

2916 **5.5.3.44 Schema (deprecated)**

2917 **DEPRECATED**

2918 The Schema string qualifier is deprecated. The schema for any feature can be determined by examining
2919 the complete class name of the class defining that feature.

2920 The Schema string qualifier takes string values, has Scope (Property, Method) and has Flavor
2921 (DisableOverride, Translatable). The default value is NULL.

2922 The Schema qualifier indicates the name of the schema that contains the feature.

2923 **DEPRECATED**

2924 **5.5.3.45 Source (removed)**

2925 This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0
2926 of this document.

2927 **5.5.3.46 SourceType (removed)**

2928 This instance-level qualifier and the corresponding pragma were removed as an erratum in version 2.3.0
2929 of this document.

2930 **5.5.3.47 Static**

2931 The Static qualifier takes boolean values, has Scope (Property, Method) and has Flavor
2932 (DisableOverride). The default value is FALSE.

2933 The property or method is static. For a definition of static properties, see 7.5.5. For a definition of static
2934 methods, see 7.9.1.

2935 An element that overrides a non-static element shall not be a static element.

2936 **5.5.3.48 Terminal**

2937 The Terminal qualifier takes boolean values, has Scope (Class, Association, Indication) and has Flavor
2938 (EnableOverride). The default value is FALSE.

2939 The class can have no subclasses. If such a subclass is declared, the compiler generates an error.

2940 This qualifier cannot coexist with the Abstract qualifier. If both are specified, the compiler generates an
2941 error.

2942 **5.5.3.49 UMLPackagePath**

2943 The UMLPackagePath qualifier takes string values, has Scope (Class, Association, Indication) and has
2944 Flavor (EnableOverride). The default value is NULL.

2945 This qualifier specifies a position within a UML package hierarchy for a CIM class.

2946 The qualifier value shall consist of a series of package names, each interpreted as a package within the
 2947 preceding package, separated by '::'. The first package name in the qualifier value shall be the schema
 2948 name of the qualified CIM class.

2949 For example, consider a class named "CIM_Abc" that is in a package named "PackageB" that is in a
 2950 package named "PackageA" that, in turn, is in a package named "CIM." The resulting qualifier
 2951 specification for this class "CIM_Abc" is as follows:

2952 `UMLPACKAGEPATH ("CIM::PackageA::PackageB")`

2953 A value of NULL indicates that the following default rule shall be used to create the UML package path:
 2954 The name of the UML package path is the schema name of the class, followed by "::default".

2955 For example, a class named "CIM_Xyz" with a UMLPackagePath qualifier value of NULL has the UML
 2956 package path "CIM::default".

2957 **5.5.3.50 Units (deprecated)**

2958 **DEPRECATED**

2959 The Units qualifier is deprecated. Instead, the PUnit qualifier should be used for programmatic access,
 2960 and the CIM client should use its own conventions to construct a string to be displayed from the PUnit
 2961 qualifier.

2962 The Units qualifier takes string values, has Scope (Property, Parameter, Method) and has Flavor
 2963 (EnableOverride, Translatable). The default value is NULL.

2964 The Units qualifier specifies the unit of measure of the qualified property, method return value, or method
 2965 parameter. For example, a Size property might have a unit of "Bytes."

2966 NULL indicates that the unit is unknown. An empty string indicates that the qualified property, method
 2967 return value, or method parameter has no unit and therefore is dimensionless. The complete set of DMTF
 2968 defined values for the Units qualifier is presented in ANNEX C.

2969 **DEPRECATED**

2970 **5.5.3.51 ValueMap**

2971 The ValueMap qualifier takes string array values, has Scope (Property, Parameter, Method) and has
 2972 Flavor (EnableOverride). The default value is NULL.

2973 The ValueMap qualifier defines the set of permissible values for the qualified property, method return, or
 2974 method parameter.

2975 The ValueMap qualifier can be used alone or in combination with the Values qualifier. When it is used
 2976 with the Values qualifier, the location of the value in the ValueMap array determines the location of the
 2977 corresponding entry in the Values array.

2978 ValueMap may be used only with string or integer types.

2979 When used with a string typed element the following rules apply:

- 2980 • a ValueMap entry shall be a string value as defined by the `stringValue` ABNF rule defined in
 2981 ANNEX A.
- 2982 • the set of ValueMap entries defined on a schema element may be extended in overriding
 2983 schema elements in subclasses or in revisions of a schema within the same major version of
 2984 the schema.

2985 When used with an integer typed element the following rules apply:

- 2986 • a ValueMap entry shall be a string value as defined by the `stringValue` ABNF rule defined in
- 2987 ANNEX A, whose string value conforms to the `integerValueMapEntry` ABNF rule:

```
2988 integerValueMapEntry = integerValue / integerValueRange
```

2989

```
2990 integerValueRange = [integerValue] ".." [integerValue]
```

2991 Where `integerValue` is defined in ANNEX A.

2992 When used with an integer type, a ValueMap entry of:

2993 `"x"` claims the value `x`.

2994 `". . x"` claims all values less than and including `x`.

2995 `"x . ."` claims all values greater than and including `x`.

2996 `". ."` claims all values not otherwise claimed.

2997 The values claimed are constrained by the value range of the data type of the qualified schema element.

2998 The usage of `". ."` as the only entry in the ValueMap array is not permitted.

2999 If the ValueMap qualifier is used together with the Values qualifier, then all values claimed by a particular
3000 ValueMap entry apply to the corresponding Values entry.

3001 EXAMPLE:

```
3002 [Values {"zero&one", "2to40", "fifty", "the unclaimed", "128-255"},
```

```
3003 ValueMap {"..1", "2..40" "50", "..", "x80.." }]
```

```
3004 uint8 example;
```

3005 In this example, where the type is `uint8`, the following mappings are made:

3006 `"..1"` and `"zero&one"` map to 0 and 1.

3007 `"2..40"` and `"2to40"` map to 2 through 40.

3008 `".."` and `"the unclaimed"` map to 41 through 49 and to 51 through 127.

3009 `"0x80.."` and `"128-255"` map to 128 through 255.

3010 An overriding property that specifies the ValueMap qualifier shall not map any values not allowed by the
3011 overridden property. In particular, if the overridden property specifies or inherits a ValueMap qualifier,
3012 then the overriding ValueMap qualifier must map only values that are allowed by the overridden
3013 ValueMap qualifier. However, the overriding property may organize these values differently than does the
3014 overridden property. For example, `ValueMap {"0..10"}` may be overridden by `ValueMap {"0..1", "2..9"}`. An
3015 overriding ValueMap qualifier may specify fewer values than the overridden property would otherwise
3016 allow.

3017 5.5.3.52 Values

3018 The Values qualifier takes string array values, has Scope (Property, Parameter, Method) and has Flavor
3019 (EnableOverride, Translatable). The default value is NULL.

3020 The Values qualifier translates between integer values and strings (such as abbreviations or English
 3021 terms) in the ValueMap array, and an associated string at the same index in the Values array. If a
 3022 ValueMap qualifier is not present, the Values array is indexed (zero relative) using the value in the
 3023 associated property, method return type, or method parameter. If a ValueMap qualifier is present, the
 3024 Values index is defined by the location of the property value in the ValueMap. If both Values and
 3025 ValueMap are specified or inherited, the number of entries in the Values and ValueMap arrays shall
 3026 match.

3027 **5.5.3.53 Version**

3028 The Version qualifier takes string values, has Scope (Class, Association, Indication) and has Flavor
 3029 (Restricted, Translatable). The default value is NULL.

3030 The Version qualifier provides the version information of the object, which increments when changes are
 3031 made to the object.

3032 Starting with CIM Schema 2.7 (including extension schema), the Version qualifier shall be present on
 3033 each class to indicate the version of the last update to the class.

3034 The string representing the version comprises three decimal integers separated by periods; that is,
 3035 M.N.U, as defined by the following ABNF:

```
3036 versionFormat = decimalValue "." decimalValue "." decimalValue
```

3037 The meaning of M.N.U is as follows:

3038 **M** – The major version in numeric form of the change to the class.

3039 **N** – The minor version in numeric form of the change to the class.

3040 **U** – The update (for example, errata, patch, ...) in numeric form of the change to the class.

3041 NOTE 1: The addition or removal of the Experimental qualifier does not require the version information to be
 3042 updated.

3043 NOTE 2: The version change applies only to elements that are local to the class. In other words, the version change
 3044 of a superclass does not require the version in the subclass to be updated.

3045 EXAMPLES:

```
3046 Version("2.7.0")
```

```
3047  
3048 Version("1.0.0")
```

3049 **5.5.3.54 Weak**

3050 The Weak qualifier takes boolean values, has Scope (Reference) and has Flavor (DisableOverride). The
 3051 default value is FALSE.

3052 This qualifier indicates that the qualified reference is weak, rendering its owning association a weak
 3053 association.

3054 For a description of the concepts of weak associations and key propagation as well as further rules
 3055 around them, see 8.2.

3056 **5.5.3.55 Write**

3057 The Write qualifier takes boolean values, has Scope (Property) and has Flavor (EnableOverride). The
 3058 default value is FALSE.

3059 The modeling semantics of a property support modification of that property by consumers. The purpose of
3060 this qualifier is to capture modeling semantics and not to address more dynamic characteristics such as
3061 provider capability or authorization rights.

3062 **5.5.3.56 XMLNamespaceName**

3063 The XMLNamespaceName qualifier takes string values, has Scope (Property, Method, Parameter) and
3064 has Flavor (EnableOverride). The default value is NULL.

3065 The XMLNamespaceName qualifier shall be specified only on elements of type string or array of string.

3066 If the effective value of the qualifier is not NULL, this indicates that the value of the qualified element is an
3067 XML instance document. The value of the qualifier in this case shall be the namespace name of the XML
3068 schema to which the XML instance document conforms.

3069 As defined in *Namspaces in XML*, the format of the namespace name shall be that of a URI reference
3070 as defined in [RFC3986](#). Two such URI references may be equivalent even if they are not equal according
3071 to a character-by-character comparison (e.g., due to usage of URI escape characters or different lexical
3072 case).

3073 If a specification of the XMLNamespaceName qualifier overrides a non-NULL qualifier value specified on
3074 an ancestor of the qualified element, the XML schema specified on the qualified element shall be a
3075 subset or restriction of the XML schema specified on the ancestor element, such that any XML instance
3076 document that conforms to the XML schema specified on the qualified element also conforms to the XML
3077 schema specified on the ancestor element.

3078 No particular XML schema description language (e.g., W3C XML Schema as defined in [XML Schema
3079 Part 0: Primer Second Edition](#) or RELAX NG as defined in [ISO/IEC 19757-2:2008](#)) is implied by usage of
3080 this qualifier.

3081 **5.5.4 Optional Qualifiers**

3082 The following subclauses list the optional qualifiers that address situations that are not common to all
3083 CIM-compliant implementations. Thus, CIM-compliant implementations can ignore optional qualifiers
3084 because they are not required to interpret or understand them. The optional qualifiers are provided in the
3085 specification to avoid random user-defined qualifiers for these recurring situations.

3086 **5.5.4.1 Alias**

3087 The Alias qualifier takes string values, has Scope (Property, Reference, Method) and has Flavor
3088 (EnableOverride, Translatable). The default value is NULL.

3089 The Alias qualifier establishes an alternate name for a property or method in the schema.

3090 **5.5.4.2 Delete**

3091 The Delete qualifier takes boolean values, has Scope (Association, Reference) and has Flavor
3092 (EnableOverride). The default value is FALSE.

3093 **For associations:** The qualified association shall be deleted if any of the objects referenced in the
3094 association are deleted and the respective object referenced in the association is qualified with IfDeleted.

3095 **For references:** The referenced object shall be deleted if the association containing the reference is
3096 deleted and qualified with IfDeleted. It shall also be deleted if any objects referenced in the association
3097 are deleted and the respective object referenced in the association is qualified with IfDeleted.

3098 CIM clients shall chase associations according to the modeled semantic and delete objects appropriately.

3099 NOTE: This usage rule must be verified when the CIM security model is defined.

3100 **5.5.4.3 DisplayDescription**

3101 The DisplayDescription qualifier takes string values, has Scope (Class, Association, Indication, Property,
3102 Reference, Parameter, Method) and has Flavor (EnableOverride, Translatable). The default value is
3103 NULL.

3104 The DisplayDescription qualifier defines descriptive text for the qualified element for display on a human
3105 interface — for example, fly-over Help or field Help.

3106 The DisplayDescription qualifier is for use within extension subclasses of the CIM schema to provide
3107 display descriptions that conform to the information development standards of the implementing product.
3108 A value of NULL indicates that no display description is provided. Therefore, a display description
3109 provided by the corresponding schema element of a superclass can be removed without substitution.

3110 **5.5.4.4 Expensive**

3111 The Expensive qualifier takes boolean values, has Scope (Class, Association, Indication, Property,
3112 Reference, Parameter, Method) and has Flavor (EnableOverride). The default value is FALSE.

3113 The Expensive qualifier indicates that the element is expensive to manipulate and/or compute.

3114 **5.5.4.5 IfDeleted**

3115 The IfDeleted qualifier takes boolean values, has Scope (Association, Reference) and has Flavor
3116 (EnableOverride). The default value is FALSE.

3117 All objects qualified by Delete within the association shall be deleted if the referenced object or the
3118 association, respectively, is deleted.

3119 **5.5.4.6 Invisible**

3120 The Invisible qualifier takes boolean values, has Scope (Class, Association, Property, Reference,
3121 Method) and has Flavor (EnableOverride). The default value is FALSE.

3122 The Invisible qualifier indicates that the element is defined only for internal purposes and should not be
3123 displayed or otherwise relied upon. For example, an intermediate value in a calculation or a value to
3124 facilitate association semantics is defined only for internal purposes.

3125 **5.5.4.7 Large**

3126 The Large qualifier takes boolean values, has Scope (Class, Property) and has Flavor (EnableOverride).
3127 The default value is FALSE.

3128 The Large qualifier property or class requires a large amount of storage space.

3129 **5.5.4.8 PropertyUsage**

3130 The PropertyUsage qualifier takes string values, has Scope (Property) and has Flavor (EnableOverride).
3131 The default value is "CURRENTCONTEXT".

3132 This qualifier allows properties to be classified according to how they are used by managed elements.
3133 Therefore, the managed element can convey intent for property usage. The qualifier does not convey
3134 what access CIM has to the properties. That is, not all configuration properties are writeable. Some
3135 configuration properties may be maintained by the provider or resource that the managed element
3136 represents, and not by CIM. The PropertyUsage qualifier enables the programmer to distinguish between
3137 properties that represent attributes of the following:

- 3138 • A managed resource versus capabilities of a managed resource

- 3139 • Configuration data for a managed resource versus metrics about or from a managed resource
- 3140 • State information for a managed resource.

3141 If the qualifier value is set to CurrentContext (the default value), then the value of PropertyUsage should
 3142 be determined by looking at the class in which the property is placed. The rules for which default
 3143 PropertyUsage values belong to which classes/subclasses are as follows:

3144 Class>CurrentContext PropertyUsage Value
 3145 Setting > Configuration
 3146 Configuration > Configuration
 3147 Statistic > Metric ManagedSystemElement > State Product > Descriptive
 3148 FRU > Descriptive
 3149 SupportAccess > Descriptive
 3150 Collection > Descriptive

3151 The valid values for this qualifier are as follows:

- 3152 • **UNKNOWN.** The property's usage qualifier has not been determined and set.
- 3153 • **OTHER.** The property's usage is not Descriptive, Capabilities, Configuration, Metric, or State.
- 3154 • **CURRENTCONTEXT.** The PropertyUsage value shall be inferred based on the class placement
 3155 of the property according to the following rules:
 - 3156 – If the property is in a subclass of Setting or Configuration, then the PropertyUsage value of
 3157 CURRENTCONTEXT should be treated as CONFIGURATION.
 - 3158 – If the property is in a subclass of Statistics, then the PropertyUsage value of
 3159 CURRENTCONTEXT should be treated as METRIC.
 - 3160 – If the property is in a subclass of ManagedSystemElement, then the PropertyUsage value
 3161 of CURRENTCONTEXT should be treated as STATE.
 - 3162 – If the property is in a subclass of Product, FRU, SupportAccess or Collection, then the
 3163 PropertyUsage value of CURRENTCONTEXT should be treated as DESCRIPTIVE.
- 3164 • **DESCRIPTIVE.** The property contains information that describes the managed element, such
 3165 as vendor, description, caption, and so on. These properties are generally not good candidates
 3166 for representation in Settings subclasses.
- 3167 • **CAPABILITY.** The property contains information that reflects the inherent capabilities of the
 3168 managed element regardless of its configuration. These are usually specifications of a product.
 3169 For example, VideoController.MaxMemorySupported=128 is a capability.
- 3170 • **CONFIGURATION.** The property contains information that influences or reflects the
 3171 configuration state of the managed element. These properties are candidates for representation
 3172 in Settings subclasses. For example, VideoController.CurrentRefreshRate is a configuration
 3173 value.
- 3174 • **STATE** indicates that the property contains information that reflects or can be used to derive the
 3175 current status of the managed element.
- 3176 • **METRIC** indicates that the property contains a numerical value representing a statistic or metric
 3177 that reports performance-oriented and/or accounting-oriented information for the managed
 3178 element. This would be appropriate for properties containing counters such as
 3179 "BytesProcessed".

3180 5.5.4.9 Provider

3181 The Provider qualifier takes string values, has Scope (Class, Association, Indication, Property, Reference,
3182 Parameter, Method) and has Flavor (EnableOverride). The default value is NULL.

3183 An implementation-specific handle to a class implementation within a CIM server.

3184 5.5.4.10 Syntax

3185 The Syntax qualifier takes string values, has Scope (Property, Reference, Parameter, Method) and has
3186 Flavor (EnableOverride). The default value is NULL.

3187 The Syntax qualifier indicates the specific type assigned to a data item. It must be used with the
3188 SyntaxType qualifier.

3189 5.5.4.11 SyntaxType

3190 The SyntaxType qualifier takes string values, has Scope (Property, Reference, Parameter, Method) and
3191 has Flavor (EnableOverride). The default value is NULL.

3192 The SyntaxType qualifier defines the format of the Syntax qualifier. It must be used with the Syntax
3193 qualifier.

3194 5.5.4.12 TriggerType

3195 The TriggerType qualifier takes string values, has Scope (Class, Association, Indication, Property,
3196 Reference, Method) and has Flavor (EnableOverride). The default value is NULL.

3197 The TriggerType qualifier specifies the circumstances that cause a trigger to be fired.

3198 The trigger types vary by meta-model construct. For classes and associations, the legal values are
3199 CREATE, DELETE, UPDATE, and ACCESS. For properties and references, the legal values are
3200 UPDATE and ACCESS. For methods, the legal values are BEFORE and AFTER. For indications, the
3201 legal value is THROWN.

3202 5.5.4.13 UnknownValues

3203 The UnknownValues qualifier takes string array values, has Scope (Property) and has Flavor
3204 (DisableOverride). The default value is NULL.

3205 The UnknownValues qualifier specifies a set of values that indicates that the value of the associated
3206 property is unknown. Therefore, the property cannot be considered to have a valid or meaningful value.

3207 The conventions and restrictions for defining unknown values are the same as those for the ValueMap
3208 qualifier.

3209 The UnknownValues qualifier cannot be overridden because it is unreasonable for a subclass to treat as
3210 known a value that a superclass treats as unknown.

3211 5.5.4.14 UnsupportedValues

3212 The UnsupportedValues qualifier takes string array values, has Scope (Property) and has Flavor
3213 (DisableOverride). The default value is NULL.

3214 The UnsupportedValues qualifier specifies a set of values that indicates that the value of the associated
3215 property is unsupported. Therefore, the property cannot be considered to have a valid or meaningful
3216 value.

3217 The conventions and restrictions for defining unsupported values are the same as those for the ValueMap
3218 qualifier.

3219 The UnsupportedValues qualifier cannot be overridden because it is unreasonable for a subclass to treat
3220 as supported a value that a superclass treats as unknown.

3221 **5.5.5 User-defined Qualifiers**

3222 The user can define any additional arbitrary named qualifiers. However, it is recommended that only
3223 defined qualifiers be used and that the list of qualifiers be extended only if there is no other way to
3224 accomplish the objective.

3225 **5.5.6 Mapping Entities of Other Information Models to CIM**

3226 The MappingStrings qualifier can be used to map entities of other information models to CIM or to
3227 express that a CIM element represents an entity of another information model. Several mapping string
3228 formats are defined in this clause to use as values for this qualifier. The CIM schema shall use only the
3229 mapping string formats defined in this document. Extension schemas should use only the mapping string
3230 formats defined in this document.

3231 The mapping string formats defined in this document conform to the following formal syntax defined in
3232 ABNF:

```
3233 mappingstrings_format = mib_format / oid_format / general_format / mif_format
```

3234 NOTE: As defined in the respective clauses, the "MIB", "OID", and "MIF" formats support a limited form of extensibility
3235 by allowing an open set of defining bodies. However, the syntax defined for these formats does not allow variations
3236 by defining body; they need to conform. A larger degree of extensibility is supported in the general format, where the
3237 defining bodies may define a part of the syntax used in the mapping.

3238 **5.5.6.1 SNMP-Related Mapping String Formats**

3239 The two SNMP-related mapping string formats, Management Information Base (MIB) and globally unique
3240 object identifier (OID), can express that a CIM element represents a MIB variable. As defined in
3241 [RFC1155](#), a MIB variable has an associated variable name that is unique within a MIB and an OID that is
3242 unique within a management protocol.

3243 The "MIB" mapping string format identifies a MIB variable using naming authority, MIB name, and variable
3244 name. It may be used only on CIM properties, parameters, or methods. The format is defined as follows,
3245 using ABNF:

```
3246 mib_format = "MIB" "." mib_naming_authority "|" mib_name "." mib_variable_name
```

3247 Where:

```
3248 mib_naming_authority = 1*(stringChar)
```

3249 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical
3250 bar (|) characters are not allowed.

```
3251 mib_name = 1*(stringChar)
```

3252 is the name of the MIB as defined by the MIB naming authority (for example, "HOST-RESOURCES-
3253 MIB"). The dot (.) and vertical bar (|) characters are not allowed.

```
3254 mib_variable_name = 1*(stringChar)
```

3255 is the name of the MIB variable as defined in the MIB (for example, "hrSystemDate"). The dot (.)
3256 and vertical bar (|) characters are not allowed.

3257 The MIB name should be the ASN.1 module name of the MIB (that is, not the RFC number). For example,
 3258 instead of using "RFC1493", the string "BRIDGE-MIB" should be used.

3259 EXAMPLE:

```
3260 [MappingStrings { "MIB.IETF|HOST-RESOURCES-MIB.hrSystemDate" }]
3261 datetime LocalDateTime;
```

3262 The "OID" mapping string format identifies a MIB variable using a management protocol and an object
 3263 identifier (OID) within the context of that protocol. This format is especially important for mapping
 3264 variables defined in private MIBs. It may be used only on CIM properties, parameters, or methods. The
 3265 format is defined as follows, using ABNF:

```
3266 oid_format = "OID" "." oid_naming_authority "|" oid_protocol_name "." oid
```

3267 Where:

```
3268 oid_naming_authority = 1*(stringChar)
```

3269 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical
 3270 bar (|) characters are not allowed.

```
3271 oid_protocol_name = 1*(stringChar)
```

3272 is the name of the protocol providing the context for the OID of the MIB variable (for example,
 3273 "SNMP"). The dot (.) and vertical bar (|) characters are not allowed.

```
3274 oid = 1*(stringChar)
```

3275 is the object identifier (OID) of the MIB variable in the context of the protocol (for example,
 3276 "1.3.6.1.2.1.25.1.2").

3277 EXAMPLE:

```
3278 [MappingStrings { "OID.IETF|SNMP.1.3.6.1.2.1.25.1.2" }]
3279 datetime LocalDateTime;
```

3280 For both mapping string formats, the name of the naming authority defining the MIB shall be one of the
 3281 following:

- 3282 • The name of a standards body (for example, IETF), for standard MIBs defined by that standards
 3283 body
- 3284 • A company name (for example, Acme), for private MIBs defined by that company

3285 5.5.6.2 General Mapping String Format

3286 This clause defines the mapping string format, which provides a basis for future mapping string formats.
 3287 Future mapping string formats defined in this document should be based on the general mapping string
 3288 format. A mapping string format based on this format shall define the kinds of CIM elements with which it
 3289 is to be used.

3290 The format is defined as follows, using ABNF. The division between the name of the format and the
 3291 actual mapping is slightly different than for the "MIF", "MIB", and "OID" formats:

```
3292 general_format = general_format_fullname "|" general_format_mapping
```

3293 Where:

```
3294 general_format_fullname = general_format_name "." general_format_defining_body
```

```
3295 general_format_name = 1*(stringChar)
```

3296 is the name of the format, unique within the defining body. The dot (.) and vertical bar (|)
3297 characters are not allowed.

```
3298 general_format_defining_body = 1*(stringChar)
```

3299 is the name of the defining body. The dot (.) and vertical bar (|) characters are not allowed.

```
3300 general_format_mapping = 1*(stringChar)
```

3301 is the mapping of the qualified CIM element, using the named format.

3302 The text in Table 8 is an example that defines a mapping string format based on the general mapping
3303 string format.

3304 **Table 8 – Example for Mapping a String Format Based on the General Mapping String Format**

General Mapping String Formats Defined for InfiniBand Trade Association (IBTA)	
IBTA defines the following mapping string formats, which are based on the general mapping string format:	
<pre>"MAD.IBTA"</pre>	
This format expresses that a CIM element represents an IBTA MAD attribute. It shall be used only on CIM properties, parameters, or methods. It is based on the general mapping string format as follows, using ABNF:	
<pre>general_format_fullname = "MAD" "." "IBTA"</pre>	
<pre>general_format_mapping = mad_class_name " " mad_attribute_name</pre>	
Where:	
<pre>mad_class_name = 1*(stringChar)</pre>	
is the name of the MAD class. The dot (.) and vertical bar () characters are not allowed.	
<pre>mad_attribute_name = 1*(stringChar)</pre>	
is the name of the MAD attribute, which is unique within the MAD class. The dot (.) and vertical bar () characters are not allowed.	

3305 **5.5.6.3 MIF-Related Mapping String Format**

3306 Management Information Format (MIF) attributes can be mapped to CIM elements using the
3307 MappingStrings qualifier. This qualifier maps DMTF and vendor-defined MIF groups to CIM classes or
3308 properties using either domain or recast mapping.

3309 **DEPRECATED**

3310 MIF is defined in the DMTF *Desktop Management Interface Specification*, which completed DMTF end of
3311 life in 2005 and is therefore no longer considered relevant. Any occurrence of the MIF format in values of
3312 the MappingStrings qualifier is considered deprecated. Any other usage of MIF in this document is also
3313 considered deprecated. The MappingStrings qualifier itself is not deprecated because it is used for
3314 formats other than MIF.

3315 **DEPRECATED**

3316 As stated in the DMTF *Desktop Management Interface Specification*, every MIF group defines a unique
3317 identification that uses the MIF class string, which has the following formal syntax defined in ABNF:

3318 `mif_class_string = mif_defining_body "|" mif_specific_name "|" mif_version`

3319 Where:

3320 `mif_defining_body = 1*(stringChar)`

3321 is the name of the body defining the group. The dot (.) and vertical bar (|) characters are not
3322 allowed.

3323 `mif_specific_name = 1*(stringChar)`

3324 is the unique name of the group. The dot (.) and vertical bar (|) characters are not allowed.

3325 `mif_version = 3(decimalDigit)`

3326 is a three-digit number that identifies the version of the group definition.

3327 The DMTF *Desktop Management Interface Specification* considers MIF class strings to be opaque
3328 identification strings for MIF groups. MIF class strings that differ only in whitespace characters are
3329 considered to be different identification strings.

3330 In addition, each MIF attribute has a unique numeric identifier, starting with the number one, using the
3331 following formal syntax defined in ABNF:

3332 `mif_attribute_id = positiveDecimalDigit *decimalDigit`

3333 A MIF domain mapping maps an individual MIF attribute to a particular CIM property. A MIF recast
3334 mapping maps an entire MIF group to a particular CIM class.

3335 The MIF format for use as a value of the MappingStrings qualifier has the following formal syntax defined
3336 in ABNF:

3337 `mif_format = mif_attribute_format | mif_group_format`

3338 Where:

3339 `mif_attribute_format = "MIF" "." mif_class_string "." mif_attribute_id`

3340 is used for mapping a MIF attribute to a CIM property.

3341 `mif_group_format = "MIF" "." mif_class_string`

3342 is used for mapping a MIF group to a CIM class.

3343 For example, a MIF domain mapping of a MIF attribute to a CIM property is as follows:

3344 `[MappingStrings { "MIF.DMTF|ComponentID|001.4" }]`
3345 `string SerialNumber;`

3346 A MIF recast mapping maps an entire MIF group into a CIM class, as follows:

3347 `[MappingStrings { "MIF.DMTF|Software Signature|002" }]`
3348 `class SoftwareSignature`
3349 `{`
3350 `...`
3351 `};`

3352 **6 Managed Object Format**

3353 The management information is described in a language based on ISO/IEC 14750:1999 called the
3354 Managed Object Format (MOF). In this document, the term "MOF specification" refers to a collection of
3355 management information described in a way that conforms to the MOF syntax. Elements of MOF syntax
3356 are introduced on a case-by-case basis with examples. In addition, a complete description of the MOF
3357 syntax is provided in ANNEX A.

3358 The MOF syntax describes object definitions in textual form and therefore establishes the syntax for
3359 writing definitions. The main components of a MOF specification are textual descriptions of classes,
3360 associations, properties, references, methods, and instance declarations and their associated qualifiers.
3361 Comments are permitted.

3362 In addition to serving the need for specifying the managed objects, a MOF specification can be processed
3363 using a compiler. To assist the process of compilation, a MOF specification consists of a series of
3364 compiler directives.

3365 MOF files shall be represented in Normalization Form C (NFC, defined in), and in one of the coded
3366 representation forms UTF-8, UTF-16BE or UTF-16LE (defined in [ISO/IEC 10646:2003](#)). UTF-8 is the
3367 recommended form for MOF files.

3368 MOF files represented in UTF-8 should not have a signature sequence (EF BB BF, as defined in Annex H
3369 of [ISO/IEC 10646:2003](#)).

3370 MOF files represented in UTF-16BE contain a big endian representation of the 16-bit data entities in the
3371 file; Likewise, MOF files represented in UTF-16LE contain little endian data entities. In both cases, they
3372 shall have a signature sequence (FEFF, as defined in Annex H of [ISO/IEC 10646:2003](#)).

3373 Consumers of MOF files should use the signature sequence or absence thereof to determine the coded
3374 representation form.

3375 This can be achieved by evaluating the first few Bytes in the file:

- 3376 • FE FF → UTF-16BE
- 3377 • FF FE → UTF-16LE
- 3378 • EF BB BF → UTF-8
- 3379 • otherwise → UTF-8

3380 In order to test whether the 16-bit entities in the two UTF-16 cases need to be byte-wise swapped before
3381 processing, evaluate the first 16-bit data entity as a 16-bit unsigned integer. If it evaluates to 0xFEFF,
3382 there is no need to swap, otherwise (0xFFEF), there is a need to swap.

3383 Consumers of MOF files shall ignore the UCS character the signature represents, if present.

3384 **6.1 MOF Usage**

3385 The managed object descriptions in a MOF specification can be validated against an active namespace
3386 (see clause 8). Such validation is typically implemented in an entity acting in the role of a CIM server. This
3387 clause describes the behavior of an implementation when introducing a MOF specification into a
3388 namespace. Typically, such a process validates both the syntactic correctness of a MOF specification and
3389 its semantic correctness against a particular implementation. In particular, MOF declarations must be
3390 ordered correctly with respect to the target implementation state. For example, if the specification
3391 references a class without first defining it, the reference is valid only if the CIM server already has a
3392 definition of that class. A MOF specification can be validated for the syntactic correctness alone, in a
3393 component such as a MOF compiler.

3394 6.2 Class Declarations

3395 A class declaration is treated as an instruction to create a new class. Whether the process of introducing
3396 a MOF specification into a namespace can add classes or modify classes is a local matter. If the
3397 specification references a class without first defining it, the CIM server must reject it as invalid if it does
3398 not already have a definition of that class.

3399 6.3 Instance Declarations

3400 Any instance declaration is treated as an instruction to create a new instance where the key values of the
3401 object do not already exist or an instruction to modify an existing instance where an object with identical
3402 key values already exists.

3403 7 MOF Components

3404 The following subclauses describe the components of MOF syntax.

3405 7.1 Keywords

3406 All keywords in the MOF syntax are case-insensitive.

3407 7.2 Comments

3408 Comments may appear anywhere in MOF syntax and are indicated by either a leading double slash (//)
3409 or a pair of matching /* and */ sequences.

3410 A // comment is terminated by carriage return, line feed, or the end of the MOF specification (whichever
3411 comes first).

3412 EXAMPLE:

```
3413 // This is a comment
```

3414 A /* comment is terminated by the next */ sequence or by the end of the MOF specification (whichever
3415 comes first). The meta model does not recognize comments, so they are not preserved across
3416 compilations. Therefore, the output of a MOF compilation is not required to include any comments.

3417 7.3 Validation Context

3418 Semantic validation of a MOF specification involves an explicit or implied namespace context. This is
3419 defined as the namespace against which the objects in the MOF specification are validated and the
3420 namespace in which they are created. Multiple namespaces typically indicate the presence of multiple
3421 management spaces or multiple devices.

3422 7.4 Naming of Schema Elements

3423 This clause describes the rules for naming schema elements, including classes, properties, qualifiers,
3424 methods, and namespaces.

3425 CIM is a conceptual model that is not bound to a particular implementation. Therefore, it can be used to
3426 exchange management information in a variety of ways, examples of which are described in the
3427 Introduction clause. Some implementations may use case-sensitive technologies, while others may use
3428 case-insensitive technologies. The naming rules defined in this clause allow efficient implementation in
3429 either environment and enable the effective exchange of management information among all compliant
3430 implementations.

3431 All names are case-insensitive, so two schema item names are identical if they differ only in case. This is
3432 mandated so that scripting technologies that are case-insensitive can leverage CIM technology. However,
3433 string values assigned to properties and qualifiers are not covered by this rule and must be treated as
3434 case-sensitive.

3435 The case of a name is set by its defining occurrence and must be preserved by all implementations. This
3436 is mandated so that implementations can be built using case-sensitive technologies such as Java and
3437 object databases. This also allows names to be consistently displayed using the same user-friendly
3438 mixed-case format. For example, an implementation, if asked to create a Disk class must reject the
3439 request if there is already a DISK class in the current schema. Otherwise, when returning the name of the
3440 Disk class it must return the name in mixed case as it was originally specified.

3441 CIM does not currently require support for any particular query language. It is assumed that
3442 implementations will specify which query languages are supported by the implementation and will adhere
3443 to the case conventions that prevail in the specified language. That is, if the query language is case-
3444 insensitive, statements in the language will behave in a case-insensitive way.

3445 For the full rules for schema element names, see ANNEX A.

3446 **7.5 Class Declarations**

3447 A class is an object describing a grouping of data items that are conceptually related and that model an
3448 object. Class definitions provide a type system for instance construction.

3449 **7.5.1 Declaring a Class**

3450 A class is declared by specifying these components:

- 3451 • Qualifiers of the class, which can be empty, or a list of qualifier name/value bindings separated
3452 by commas (,) and enclosed with square brackets ([and]).
- 3453 • Class name.
- 3454 • Name of the class from which this class is derived, if any.
- 3455 • Class properties, which define the data members of the class. A property may also have an
3456 optional qualifier list expressed in the same way as the class qualifier list. In addition, a property
3457 has a data type, and (optionally) a default (initializer) value.
- 3458 • Methods supported by the class. A method may have an optional qualifier list, and it has a
3459 signature consisting of its return type plus its parameters and their type and usage.
- 3460 • A CIM class may expose more than one element (property or method) with a given name, but it
3461 is not permitted to define more than one element with a particular name. This can happen if a
3462 base class defines an element with the same name as an element defined in a derived class
3463 without overriding the base class element. (Although considered rare, this could happen in a
3464 class defined in a vendor extension schema that defines a property or method that uses the
3465 same name that is later chosen by an addition to an ancestor class defined in the common
3466 schema.)

3467 This sample shows how to declare a class:

```
3468 [abstract]
3469 class Win32_LogicalDisk
3470 {
3471     [read]
3472     string DriveLetter;
3473
3474     [read, Units("KiloBytes")]
```

```

3475     sint32 RawCapacity = 0;
3476
3477         [write]
3478     string VolumeLabel;
3479
3480         [Dangerous]
3481     boolean Format([in] boolean FastFormat);
3482 };

```

3483 7.5.2 Subclasses

3484 To indicate that a class is a subclass of another class, the derived class is declared by using a colon
 3485 followed by the superclass name. For example, if the class ACME_Disk_v1 is derived from the class
 3486 CIM_Media:

```

3487 class ACME_Disk_v1 : CIM_Media
3488 {
3489     // Body of class definition here ...
3490 };

```

3491 The terms base class, superclass, and supertype are used interchangeably, as are derived class,
 3492 subclass, and subtype. The superclass declaration must appear at a prior point in the MOF specification
 3493 or already be a registered class definition in the namespace in which the derived class is defined.

3494 7.5.3 Default Property Values

3495 Any properties (including references) in a class definition may have default values defined. The default
 3496 value of a property represents an initialization constraint for the property and propagates to subclasses;
 3497 for details see 5.1.2.8.

3498 The format for the specification of a default value in CIM MOF depends on the property data type, and
 3499 shall be:

- 3500 • For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A.
- 3501 • For the char16 datatype, as defined by the `charValue` or `integerValue` ABNF rules defined
 3502 in ANNEX A.
- 3503 • For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4.
 3504 Since this is a string, it may be specified in multiple pieces, as defined by the `stringValue`
 3505 ABNF rule defined in ANNEX A.
- 3506 • For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- 3507 • For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.

3508 For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.

- 3509 • For <classname> REF datatypes, the string representation of the instance path as described in
 3510 8.5.

3511 In addition, NULL may be specified as a default value for any data type.

3512 EXAMPLE:

```

3513 class ACME_Disk
3514 {
3515     string Manufacturer = "Acme";

```

```
3516     string ModelNumber = "123-AAL";
3517 };
```

3518 As defined in 7.8.2, arrays can be defined to be of type Bag, Ordered, or Indexed. For any of these array
3519 types, a default value for the array may be specified by specifying the values of the array elements in a
3520 comma-separated list delimited with curly brackets, as defined in the `arrayInitializer` ABNF rule in
3521 ANNEX A.

3522 EXAMPLE:

```
3523 class ACME_ExampleClass
3524 {
3525     [ArrayType ("Indexed")]
3526     string ip_addresses [] = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };
3527     // This variable length array has three elements as a default.
3528
3529     sint32 sint32_values [10] = { 1, 2, 3, 5, 6 };
3530     // Since fixed arrays always have their defined number
3531     // of elements, default value defines a default value of NULL
3532     // for the remaining elements.
3533 };
```

3534 7.5.4 Key Properties

3535 Instances of a class can be identified within a namespace. Designating one or more properties with the
3536 Key qualifier provides for such instance identification. For example, this class has one property (Volume)
3537 that serves as its key:

```
3538 class ACME_Drive
3539 {
3540     [Key]
3541     string Volume;
3542
3543     string FileSystem;
3544
3545     sint32 Capacity;
3546 };
```

3547 The designation of a property as a key is inherited by subclasses of the class that specified the Key
3548 qualifier on the property. For example, the `ACME_Modem` class in the following example which
3549 subclasses the `ACME_LogicalDevice` class from the previous example, has the same two key properties
3550 as its superclass:

```
3551 class ACME_Modem : ACME_LogicalDevice
3552 {
3553     uint32 ActualSpeed;
3554 };
```

3555 A subclass that inherits key properties shall not designate additional properties as keys (by specifying the
3556 Key qualifier on them) and it shall not remove the designation as a key from any inherited key properties
3557 (by specifying the Key qualifier with a value of FALSE on them).

3558 Any non-abstract class shall expose key properties.

3559 **7.5.5 Static Properties**

3560 If a property is declared as a static property, it has the same value for all CIM instances that have the
 3561 property in the same namespace. Therefore, any change in the value of a static property for a CIM
 3562 instance also affects the value of that property for the other CIM instances that have it. As for any
 3563 property, a change in the value of a static property of a CIM instance in one namespace may or may not
 3564 affect its value in CIM instances in other namespaces.

3565 Overrides on static properties are prohibited. Overrides of static methods are allowed.

3566 **7.6 Association Declarations**

3567 An association is a special kind of class describing a link between other classes. Associations also
 3568 provide a type system for instance constructions. Associations are just like other classes with a few
 3569 additional semantics, which are explained in the following subclauses.

3570 **7.6.1 Declaring an Association**

3571 An association is declared by specifying these components:

- 3572 • Qualifiers of the association (at least the Association qualifier, if it does not have a supertype).
 3573 Further qualifiers may be specified as a list of qualifier/name bindings separated by commas
 3574 (,). The entire qualifier list is enclosed in square brackets ([and]).
- 3575 • Association name. The name of the association from which this association derives (if any).
- 3576 • Association references. Define pointers to other objects linked by this association. References
 3577 may also have qualifier lists that are expressed in the same way as the association qualifier list
 3578 — especially the qualifiers to specify cardinalities of references (see 5.1.2.14). In addition, a
 3579 reference has a data type, and (optionally) a default (initializer) value.
- 3580 • Additional association properties that define further data members of this association. They are
 3581 defined in the same way as for ordinary classes.
- 3582 • The methods supported by the association. They are defined in the same way as for ordinary
 3583 classes.

3584 **EXAMPLE:** The following example shows how to declare an association (assuming given classes CIM_A and
 3585 CIM_B):

```
3586 [Association]
3587 class CIM_LinkBetweenAandB : CIM_Dependency
3588 {
3589     [Override ("Antecedent")]
3590     CIM_A REF Antecedent;
3591
3592     [Override ("Dependent")]
3593     CIM_B REF Dependent;
3594 };
```

3595 **7.6.2 Subassociations**

3596 To indicate a subassociation of another association, the same notation as for ordinary classes is used.
 3597 The derived association is declared using a colon followed by the superassociation name. (An example is
 3598 provided in 7.6.1).

3599 7.6.3 Key References and Properties in Associations

3600 Instances of an association class also can be identified within a namespace, because associations are
3601 just a special kind of a class. Designating one or more references or properties with the Key qualifier
3602 provides for such instance identification.

3603 For example, this association class designates both of its references as keys:

```
3604 [Association, Aggregation]
3605 class ACME_Component
3606 {
3607     [Aggregate, Key]
3608     ACME_ManagedSystemElement REF GroupComponent;
3609
3610     [Key]
3611     ACME_ManagedSystemElement REF PartComponent;
3612 };
```

3613 The key definition for associations follows the same rules as for ordinary classes: Compound keys are
3614 supported in the same way; keys are inherited by subassociations; Subassociations shall not add or
3615 remove keys.

3616 These rules imply that associations may designate ordinary properties (i.e., properties that are not
3617 references) as keys and that associations may designate only a subset of its references as keys.

3618 7.6.4 Weak Associations and Propagated Keys

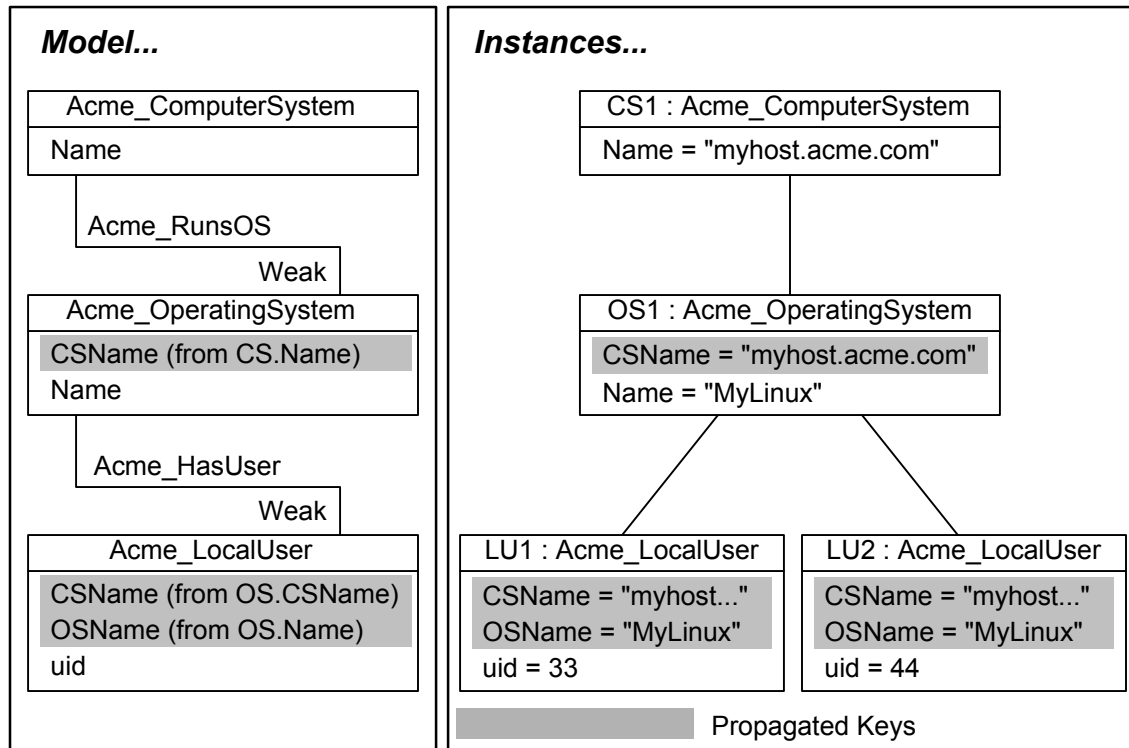
3619 CIM provides a mechanism to identify instances within the context of other associated instances. The
3620 class providing such context is called a *scoping class*, the class whose instances are identified within the
3621 context of the scoping class is called a *weak class*, and the association establishing the relation between
3622 these classes is called a *weak association*. Similarly, the instances of a scoping class are referred to as
3623 *scoping instances*, and the instances of a weak class are referred to as *weak instances*.

3624 This mechanism allows weak instances to be identifiable in a global scope even though its own key
3625 properties do not provide such uniqueness on their own. The remaining keys come from the scoping
3626 class and provide the necessary context. These keys are called *propagated keys*, because they are
3627 propagated from the scoping instance to the weak instance.

3628 A class that is not weak with respect to any other class (i.e., no references to that class are marked as
3629 weak) is referred to as a *top-level class*. More generally, a class is a top-level class if it exposes only keys
3630 that are not propagated keys.

3631 An association is designated to be a weak association by qualifying the reference to the weak class with
3632 the Weak qualifier, as defined in 5.5.3.54. The propagated keys in the weak class are designated to be
3633 propagated by qualifying them with the Propagated qualifier, as defined in 5.5.3.38.

3634 Figure 3 shows an example with two weak associations. There are three classes:
3635 ACME_ComputerSystem, ACME_OperatingSystem and ACME_LocalUser. ACME_OperatingSystem is
3636 weak with respect to ACME_ComputerSystem because the ACME_RunningOS association is marked as
3637 weak on its reference to ACME_OperatingSystem. Similarly, ACME_LocalUser is weak with respect to
3638 ACME_OperatingSystem because the ACME_HasUser association is marked as weak on its reference to
3639 ACME_LocalUser.



3640

3641

Figure 3 – Example with Two Weak Associations and Propagated Keys

3642

The following MOF classes represent the example shown in Figure 3:

3643

3644

3645

3646

3647

3648

3649

3650

3651

3652

3653

3654

3655

3656

3657

3658

3659

3660

3661

3662

3663

```

class ACME_ComputerSystem
{
    [Key]
    string Name;
};

class ACME_OperatingSystem
{
    [Key]
    string Name;

    [Key, Propagated ("ACME_ComputerSystem.Name")]
    string CSName;
};

class ACME_LocalUser
{
    [Key]
    String uid;

    [Key, Propagated("ACME_OperatingSystem.Name")]

```

```

3664     String OSName;
3665
3666         [Key, Propagated("ACME_OperatingSystem.CSName")]
3667     String CSName;
3668 };
3669
3670     [Association]
3671 class ACME_RunningOs
3672 {
3673     [Key]
3674     ACME_ComputerSystem REF ComputerSystem;
3675
3676     [Key, Weak]
3677     ACME_OperatingSystem REF OperatingSystem;
3678 };
3679
3680     [Association]
3681 class ACME_HasUser
3682 {
3683     [Key]
3684     ACME_OperatingSystem REF OperatingSystem;
3685
3686     [Key, Weak]
3687     ACME_LocalUser REF User;
3688 };

```

3689 The following rules apply:

- 3690 • The keys of top-level classes should be sufficiently unique with respect to the scope of the
3691 managed environment. For example, if a global enterprise is to be managed, the keys of any
3692 top-level classes should be unique at least within that enterprise. In the example,
3693 ACME_ComputerSystem uses domain names for its key property Name, which provides even
3694 for global uniqueness.
- 3695 • A weak class may in turn be a scoping class for another class. In the example,
3696 ACME_OperatingSystem is scoped by ACME_ComputerSystem and scopes ACME_LocalUser.
3697 Therefore, all classes can be arranged as directed graphs with the top-level classes as their
3698 roots and the weak associations as their edges.
- 3699 • The property in the scoping instance that gets propagated does not need to be a key property.
- 3700 • The association between the weak class and the scoping class shall expose a weak reference
3701 (see 5.5.3.54 "Weak") that targets the weak class.
- 3702 • No more than one association may reference a weak class with a weak reference.
- 3703 • An association may expose no more than one weak reference.
- 3704 • Key properties may propagate across multiple weak associations. In the example, property
3705 Name in the ACME_ComputerSystem class is first propagated into class
3706 ACME_OperatingSystem as property CSName, and then from there into class
3707 ACME_LocalUser as property CSName (not changing its name this time). Still, only
3708 ACME_OperatingSystem is considered the scoping class for ACME_LocalUser.

3709 NOTE: Since a reference to an instance always includes key values for the keys exposed by the class, a reference to
3710 an instance of a weak class includes the propagated keys of that class.

3711 7.6.5 Object References

3712 Object references are special properties whose values are links or pointers to other objects that are
 3713 classes or instances. The value of an object reference is the string representation of an object path, as
 3714 defined in 8.2. Consequently, the actual string value depends on the context the object reference is used
 3715 in. For example, when used in the context of a particular protocol, the string value is the string
 3716 representation defined for that protocol; when used in CIM MOF, the string value is the string
 3717 representation of object paths for CIM MOF as defined in 8.5.

3718 The data type of an object reference is declared as "XXX Ref", indicating a strongly typed reference to
 3719 objects (instances or classes) of the class with name "XXX" or a subclass of this class. Object references
 3720 in associations shall reference instances only and shall not have the special NULL value. Object
 3721 references in method parameters shall reference instances or classes or both. Only associations may
 3722 define references, ordinary classes and indications shall not define references, as defined in 5.1.2.13.

3723 EXAMPLE 1:

```
3724 [Association]
3725 class ACME_ExampleAssoc
3726 {
3727     ACME_AnotherClass REF Inst1;
3728     ACME_Aclass         REF Inst2;
3729 };
```

3730 In this declaration, Inst1 can be set to point only to instances of type ACME_AnotherClass, including
 3731 instances of its subclasses.

3732 EXAMPLE 2:

```
3733 class ACME_Modem
3734 {
3735     uint32 UseSettingsOf (
3736         ACME_Modem REF OtherModem // references an instance object
3737     );
3738 };
```

3739 In this method, parameter OtherModem is used to reference an instance object.

3740 EXAMPLE 3:

```
3741 class ACME_PolicyActivationService
3742 {
3743     uint32 ActivatePolicyClass (
3744         ACME_Policy REF PolicyClass // references a class object
3745     );
3746 };
```

3747 In this method, parameter PolicyClass is used to reference a class object. The distinction between
 3748 referencing class or instance objects is not formally declared in the reference type.

3749 The initialization of object references in association instances with object reference constants or aliases is
 3750 defined in 7.8.

3751 In associations, object references have cardinalities that are denoted using the Min and Max qualifiers.
 3752 Examples of UML cardinality notations and their respective combinations of Min and Max values are
 3753 shown in Table 9.

3754

Table 9 – UML Cardinality Notations

UML	MIN	MAX	Required MOF Text*	Description
*	0	NULL		Many
1..*	1	NULL	Min(1)	At least one
1	1	1	Min(1), Max(1)	One
0,1 (or 0..1)	0	1	Max(1)	At most one

3755 7.7 Qualifiers

3756 Qualifiers are named and typed values that provide information about CIM elements. Since the qualifier
3757 values are on CIM elements and not on CIM instances, they are considered to be meta-data.

3758 This subclause describes how qualifiers are defined in MOF. For a description of the concept of qualifiers,
3759 see 5.5.1.

3760 7.7.1 Qualifier Type

3761 As defined in 5.5.1.2, the declaration of a qualifier type allows the definition of its name, data type, scope,
3762 flavor and default value.

3763 The declaration of a qualifier type shall follow the formal syntax defined by the `qualifierDeclaration`
3764 ABNF rule defined in ANNEX A.

3765 EXAMPLE 1:

3766 The MaxLen qualifier which defines the maximum length of the string typed qualified element is declared
3767 as follows:

```
3768 qualifier MaxLen : uint32 = null,  
3769     scope (Property, Method, Parameter);
```

3770 This declaration establishes a qualifier named "MaxLen" that has a data type `uint32` and can therefore
3771 specify length values between 0 and $2^{32}-1$. It has scope (Property Method Parameter) and can therefore
3772 be specified on ordinary properties, method parameters and methods. It has no flavor specified, so it has
3773 the default flavor (ToSubclass EnableOverride) and therefore propagates to subclasses and is permitted
3774 to be overridden there. Its default value is NULL.

3775 EXAMPLE 2:

3776 The Deprecated qualifier which indicates that the qualified element is deprecated and allows the
3777 specification of replacement elements is declared as follows:

```
3778 qualifier Deprecated : string[],  
3779     scope (Any),  
3780     flavor (Restricted);
```

3781 This declaration establishes a qualifier named "Deprecated" that has a data type of array of string. It has
3782 scope (Any) and can therefore be defined on ordinary classes, associations, indications, ordinary
3783 properties, references, methods and method parameters. It has flavor (Restricted) and therefore does not
3784 propagate to subclasses. It has no default value defined, so its implied default value is NULL.

3785 7.7.2 Qualifier Value

3786 As defined in 5.5.1.1, the specification of a qualifier defines a value for that qualifier on the qualified CIM
3787 element.

3788 The specification of a set of qualifiers for a CIM element shall follow the formal syntax defined by the
3789 `qualifierList` ABNF rule defined in ANNEX A.

3790 As defined there, specification of the `qualifierList` syntax element is optional, and if specified it shall
3791 be placed before the declaration of the CIM element the qualifiers apply to.

3792 A specification of a qualifier in MOF requires that its qualifier type declaration be placed before the first
3793 specification of the qualifier on a CIM element.

3794 EXAMPLE 1:

```
3795 // Some qualifier type declarations
3796
3797 qualifier Abstract : boolean = false,
3798     scope (Class, Association, Indication),
3799     flavor (Restricted);
3800
3801 qualifier Description : string = null,
3802     scope (Any),
3803     flavor (ToSubclass, EnableOverride, Translatable);
3804
3805 qualifier MaxLen : uint32 = null,
3806     scope (Property, Method, Parameter),
3807     flavor (ToSubclass, EnableOverride);
3808
3809 qualifier ValueMap : string[],
3810     scope (Property, Method, Parameter),
3811     flavor (ToSubclass, EnableOverride);
3812
3813 qualifier Values : string[],
3814     scope (Property, Method, Parameter),
3815     flavor (ToSubclass, EnableOverride, Translatable);
3816
3817 // ...
3818
3819 // A class specifying these qualifiers
3820
3821     [Abstract (true), Description (
3822         "A system.\n"
3823         "Details are defined in subclasses.")]
3824 class ACME_System
3825 {
3826     [MaxLen (80)]
3827     string Name;
3828
3829     [ValueMap {"0", "1", "2", "3", "4..65535"},
3830     Values {"Not Applicable", "Unknown", "Other",
3831         "General Purpose", "Switch", "DMTF Reserved"}]
3832     uint16 Type;
3833 };
```

3834 In this example, the following qualifier values are specified:

- 3835 • On class ACME_System:
 - 3836 – A value of True for the Abstract qualifier
 - 3837 – A value of "A system.\nDetails are defined in subclasses." for the Description qualifier
- 3838 • On property Name:
 - 3839 – A value of 80 for the MaxLen qualifier
- 3840 • On property Type:
 - 3841 – A specific array of values for the ValueMap qualifier
 - 3842 – A specific array of values for the Values qualifier

3843 As defined in 5.5.1.5, these CIM elements do have implied values for all qualifiers that are not specified
 3844 but for which qualifier type declarations exist. Therefore, the following qualifier values are implied in
 3845 addition in this example:

- 3846 • On property Name:
 - 3847 – A value of Null for the Description qualifier
 - 3848 – An empty array for the ValueMap qualifier
 - 3849 – An empty array for the Values qualifier
- 3850 • On property Type:
 - 3851 – A value of Null for the Description qualifier

3852 Qualifiers may be specified without specifying a value. In this case, a default value is implied for the
 3853 qualifier. The implied default value depends on the data type of the qualifier, as follows:

- 3854 • For data type boolean, the implied default value is True
- 3855 • For numeric data types, the implied default value is Null
- 3856 • For string and char16 data types, the implied default value is Null
- 3857 • For arrays of any data type, the implied default is that the array is empty.

3858 EXAMPLE 2 (assuming the qualifier type declarations from example 1 in this subclause):

```
3859 [Abstract]
3860 class ACME_Device
3861 {
3862     // ...
3863 };
```

3864 In this example, the Abstract qualifier is specified without a value, therefore a value of True is implied on
 3865 this boolean typed qualifier.

3866 The concept of implying default values for qualifiers that are specified without a value is different from the
 3867 concept of using the default values defined in the qualifier type declaration. The difference is that the
 3868 latter is used when the qualifier is not specified. Consider the following example:

3869 EXAMPLE 3 (assuming the declarations from examples 1 and 2 in this subclause):

```
3870 class ACME_LogicalDevice : ACME_Device
3871 {
3872     // ...
3873 };
```

3874 In this example, the Abstract qualifier is not specified, so its effective value is determined as defined in
3875 5.5.1.5: Since the Abstract qualifier has flavor (Restricted), its effective value for class
3876 ACME_LogicalDevice is the default value defined in its qualifier type declaration, i.e., False, regardless of
3877 the value of True the Abstract qualifier has for class ACME_Device.

3878 7.8 Instance Declarations

3879 Instances are declared using the keyword sequence "instance of" and the class name. The property
3880 values of the instance may be initialized within an initialization block. Any qualifiers specified for the
3881 instance shall already be present in the defining class and shall have the same value and flavors.

3882 Property initialization consists of an optional list of preceding qualifiers, the name of the property, and an
3883 optional value which defines the default value for the property as defined in 7.5.3. Any qualifiers specified
3884 for the property shall already be present in the property definition from the defining class, and they shall
3885 have the same value and flavors.

3886 The format of initializer values for properties in instance declarations in CIM MOF depends on the data
3887 type of the property, and shall be:

- 3888 • For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A.
- 3889 • For the char16 datatype, as defined by the `charValue` or `integerValue` ABNF rules defined
3890 in ANNEX A.
- 3891 • For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4.
3892 Since this is a string, it may be specified in multiple pieces, as defined by the `stringValue`
3893 ABNF rule defined in ANNEX A.
- 3894 • For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- 3895 • For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.
- 3896 • For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.
- 3897 • For <classname> REF datatypes, as defined by the `referenceInitializer` ABNF rule defined in
3898 ANNEX A. This includes both object paths and instance aliases.

3899 In addition, NULL may be specified as an initializer value for any data type.

3900 As defined in 7.8.2, arrays can be defined to be of type Bag, Ordered, or Indexed. For any of these array
3901 types, an array property can be initialized in an instance declaration by specifying the values of the array
3902 elements in a comma-separated list delimited with curly brackets, as defined in the `arrayInitializer`
3903 ABNF rule in ANNEX A.

3904 For subclasses, all properties in the superclass may have their values initialized along with the properties
3905 in the subclass.

3906 Any property values not initialized have default values as specified in the class definition, or (if no default
3907 value is specified) the special value NULL to indicate absence of value.

3908 As defined in the description of the Key qualifier, the values of all key properties must be non-NULL.

3909 As described in item 21-E of subclause 5.1, a class may have, by inheritance, more than one property
3910 with a particular name. If a property initialization has a property name that applies to more than one
3911 property in the class, the initialization applies to the property defined closest to the class of the instance.
3912 That is, the property can be located by starting at the class of the instance. If the class defines a property
3913 with the name from the initialization, then that property is initialized. Otherwise, the search is repeated
3914 from the direct superclass of the class. See ANNEX H for more information about ambiguous property
3915 and method names.

3916 For example, given the class definition:

```
3917 class ACME_LogicalDisk : CIM_Partition
3918 {
3919     [Key]
3920     string DriveLetter;
3921
3922     [Units("kilo bytes")]
3923     sint32 RawCapacity = 128000;
3924
3925     [Write]
3926     string VolumeLabel;
3927
3928     [Units("kilo bytes")]
3929     sint32 FreeSpace;
3930 };
```

3931 an instance of this class can be declared as follows:

```
3932 instance of ACME_LogicalDisk
3933 {
3934     DriveLetter = "C";
3935     VolumeLabel = "myvol";
3936 };
```

3937 The resulting instance takes these property values:

- 3938 • DriveLetter is assigned the value "C".
- 3939 • RawCapacity is assigned the default value 128000.
- 3940 • VolumeLabel is assigned the value "myvol".
- 3941 • FreeSpace is assigned the value NULL.

3942 EXAMPLE: The following is an example with array properties:

```
3943 class ACME_ExampleClass
3944 {
3945     [ArrayType ("Indexed")]
3946     string ip_addresses []; // Indexed array of variable length
3947
3948     sint32 sint32_values [10]; // Bag array of fixed length = 10
3949 };
3950
3951 instance of ACME_ExampleClass
3952 {
3953     ip_addresses = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };
3954     // This variable length array now has three elements.
3955
3956     sint32_values = { 1, 2, 3, 5, 6 };
3957     // Since fixed arrays always have their defined number
3958     // of elements, the remaining elements have the NULL value.
3959 };
```

3960 EXAMPLE: The following is an example with instances of associations:

```
3961 class ACME_Object
3962 {
3963     string Name;
3964 };
3965
3966 class ACME_Dependency
3967 {
3968     ACME_Object REF Antecedent;
3969     ACME_Object REF Dependent;
3970 };
3971
3972 instance of ACME_Dependency
3973 {
3974     Dependent = "CIM_Object.Name = \"obj1\"";
3975     Antecedent = "CIM_Object.Name = \"obj2\"";
3976 };
```

3977 7.8.1 Instance Aliasing

3978 Aliases are symbolic references to instances located elsewhere in the MOF specification. They have
3979 significance only within the MOF specification in which they are defined, and they are no longer available
3980 and have been resolved to instance paths once the MOF specification of instances has been loaded into
3981 a CIM server.

3982 An alias can be assigned to an instance using the syntax defined for the `alias` ABNF rule in ANNEX A.
3983 Such an alias can later be used within the same MOF specification as a value for an object reference
3984 property.

3985 Forward-referencing and circular aliases are permitted.

3986 EXAMPLE:

```
3987 class ACME_Node
3988 {
3989     string Color;
3990 };
```

3991 These two instances define the aliases \$BlueNode and \$RedNode:

```
3992 instance of ACME_Node as $BlueNode
3993 {
3994     Color = "blue";
3995 };
3996
3997 instance of ACME_Node as $RedNode
3998 {
3999     Color = "red";
4000 };
4001
4002 class ACME_Edge
4003 {
```

```
4004     string Color;  
4005     ACME_Node REF Node1;  
4006     ACME_Node REF Node2;  
4007 };
```

4008 These aliases \$Bluenode and \$RedNode are used in an association instance in order to reference the
4009 two instances.

```
4010 instance of ACME_Edge  
4011 {  
4012     Color = "green";  
4013     Node1 = $BlueNode;  
4014     Node2 = $RedNode;  
4015 };
```

4016 7.8.2 Arrays

4017 Arrays of any of the basic data types can be declared in the MOF specification by using square brackets
4018 after the property or parameter identifier. If there is an unsigned integer constant within the square
4019 brackets, the array is a fixed-length array and the constant indicates the size of the array; if there is
4020 nothing within the square brackets, the array is a variable-length array. Otherwise, the array definition is
4021 invalid.

4022 Fixed-length arrays always have the specified number of elements. Elements cannot be added to or
4023 deleted from fixed-length arrays, but the values of elements can be changed.

4024 Variable-length arrays have a number of elements between 0 and an implementation-defined maximum.
4025 Elements can be added to or deleted from variable-length array properties, and the values of existing
4026 elements can be changed.

4027 Element addition, deletion, or modification is defined only for array properties because array parameters
4028 are only transiently instantiated when a CIM method is invoked. For array parameters, the array is
4029 thought to be created by the CIM client for input parameters and by the CIM server for output parameters.
4030 The array is thought to be retrieved and deleted by the CIM server for input parameters and by the CIM
4031 client for output parameters.

4032 Array indexes start at 0 and have no gaps throughout the entire array, both for fixed-length and variable-
4033 length arrays. The special NULL value signifies the absence of a value for an element, not the absence of
4034 the element itself. In other words, array elements that are NULL exist in the array and have a value of
4035 NULL. They do not represent gaps in the array.

4036 Like any CIM type, an array itself may have the special NULL value to indicate absence of value.
4037 Conceptually, the value of the array itself, if not absent, is the set of its elements. An empty array (that is,
4038 an array with no elements) must be distinguishable from an array that has the special NULL value. For
4039 example, if an array contains error messages, it makes a difference to know that there are no error
4040 messages rather than to be uncertain about whether there are any error messages.

4041 The type of an array is defined by the ArrayType qualifier with values of Bag, Ordered, or Indexed. The
4042 default array type is Bag.

4043 For a Bag array type, no significance is attached to the array index other than its convenience for
4044 accessing the elements of the array. There can be no assumption that the same index returns the same
4045 element for every retrieval, even if no element of the array is changed. The only valid assumption is that a
4046 retrieval of the entire array contains all of its elements and the index can be used to enumerate the
4047 complete set of elements within the retrieved array. The Bag array type should be used in the CIM
4048 schema when the order of elements in the array does not have a meaning. There is no concept of
4049 corresponding elements between Bag arrays.

4050 For an Ordered array type, the CIM server maintains the order of elements in the array as long as no
 4051 array elements are added, deleted, or changed. Therefore, the CIM server does not honor any order of
 4052 elements presented by the CIM client when creating the array (during creation of the CIM instance for an
 4053 array property or during CIM method invocation for an input array parameter) or when modifying the
 4054 array. Instead, the CIM server itself determines the order of elements on these occasions and therefore
 4055 possibly reorders the elements. The CIM server then maintains the order it has determined during
 4056 successive retrievals of the array. However, as soon as any array elements are added, deleted, or
 4057 changed, the CIM server again determines a new order and from then on maintains that new order. For
 4058 output array parameters, the CIM server determines the order of elements and the CIM client sees the
 4059 elements in that same order upon retrieval. The Ordered array type should be used when the order of
 4060 elements in the array does have a meaning and should be controlled by the CIM server. The order the
 4061 CIM server applies is implementation-defined unless the class defines particular ordering rules.
 4062 Corresponding elements between Ordered arrays are those that are retrieved at the same index.

4063 For an Indexed array type, the array maintains the reliability of indexes so that the same index returns the
 4064 same element for successive retrievals. Therefore, particular semantics of elements at particular index
 4065 positions can be defined. For example, in a status array property, the first array element might represent
 4066 the major status and the following elements represent minor status modifications. Consequently, element
 4067 addition and deletion is not supported for this array type. The Indexed array type should be used when
 4068 the relative order of elements in the array has a meaning and should be controlled by the CIM client, and
 4069 reliability of indexes is needed. Corresponding elements between Indexed arrays are those at the same
 4070 index.

4071 The current release of CIM does not support n-dimensional arrays.

4072 Arrays of any basic data type are legal for properties. Arrays of references are not legal for properties.
 4073 Arrays must be homogeneous; arrays of mixed types are not supported. In MOF, the data type of an
 4074 array precedes the array name. Array size, if fixed-length, is declared within square brackets after the
 4075 array name. For a variable-length array, empty square brackets follow the array name.

4076 Arrays are declared using the following MOF syntax:

```
4077 class ACME_A
4078 {
4079     [Description("An indexed array of variable length"), ArrayType("Indexed")]
4080     uint8 MyIndexedArray[];
4081
4082     [Description("A bag array of fixed length")]
4083     uint8 MyBagArray[17];
4084 };
```

4085 If default values are to be provided for the array elements, this MOF syntax is used:

```
4086 class ACME_A
4087 {
4088     [Description("A bag array property of fixed length")]
4089     uint8 MyBagArray[17] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};
4090 };
```

4091 EXAMPLE: The following MOF presents further examples of Bag, Ordered, and Indexed array declarations:

```
4092 class ACME_Example
4093 {
4094     char16 Prop1[];           // Bag (default) array of chars, Variable length
4095
4096     [ArrayType ("Ordered")] // Ordered array of double-precision reals,
```

```

4097     real64 Prop2[];           // Variable length
4098
4099     [ArrayType ("Bag")]     // Bag array containing 4 32-bit signed integers
4100     sint32 Prop3[4];
4101
4102     [ArrayType ("Ordered")] // Ordered array of strings, Variable length
4103     string Prop4[] = {"an", "ordered", "list"};
4104     // Prop4 is variable length with default values defined at the
4105     // first three positions in the array
4106
4107     [ArrayType ("Indexed")] // Indexed array of 64-bit unsigned integers
4108     uint64 Prop5[];
4109 };

```

4110 7.9 Method Declarations

4111 A method is defined as an operation with a signature that consists of a possibly empty list of parameters
 4112 and a return type. There are no restrictions on the type of parameters other than they shall be a fixed- or
 4113 variable-length array of one of the data types described in 5.2. Method return types defined in MOF must
 4114 be one of the data types described in 5.2. Return types cannot be arrays but are otherwise unrestricted.

4115 Methods are expected, but not required, to return a status value indicating the result of executing the
 4116 method. Methods may use their parameters to pass arrays.

4117 Syntactically, the only thing that distinguishes a method from a property is the parameter list. The fact that
 4118 methods are expected to have side-effects is outside the scope of this document.

4119 EXAMPLE 1: In the following example, Start and Stop methods are defined on the CIM_Service class. Each method
 4120 returns an integer value:

```

4121 class CIM_Service : CIM_LogicalElement
4122 {
4123     [Key]
4124     string Name;
4125     string StartMode;
4126     boolean Started;
4127     uint32 StartService();
4128     uint32 StopService();
4129 };

```

4130 EXAMPLE 2: In the following example, a Configure method is defined on the Physical DiskDrive class. It takes a
 4131 DiskPartitionConfiguration object reference as a parameter and returns a boolean value:

```

4132 class ACME_DiskDrive : CIM_Media
4133 {
4134     sint32 BytesPerSector;
4135     sint32 Partitions;
4136     sint32 TracksPerCylinder;
4137     sint32 SectorsPerTrack;
4138     string TotalCylinders;
4139     string TotalTracks;
4140     string TotalSectors;
4141     string InterfaceType;
4142     boolean Configure([IN] DiskPartitionConfiguration REF config);

```

4143 };

4144 7.9.1 Static Methods

4145 If a method is declared as a static method, it does not depend on any per-instance data. Non-static
4146 methods are invoked in the context of an instance; for static methods, the context of a class is sufficient.
4147 Overrides on static properties are prohibited. Overrides of static methods are allowed.

4148 7.10 Compiler Directives

4149 Compiler directives are provided as the keyword "pragma" preceded by a hash (#) character and
4150 followed by a string parameter. The current standard compiler directives are listed in Table 10.

4151 **Table 10 – Standard Compiler Directives**

Compiler Directive	Interpretation
#pragma include()	Has a file name as a parameter. The file is assumed to be a MOF file. The pragma has the effect of textually inserting the contents of the include file at the point where the include pragma is encountered.
#pragma instancelocale()	Declares the locale used for instances described in a MOF file. This pragma specifies the locale when "INSTANCE OF" MOF statements include string or char16 properties and the locale is not the same as the locale specified by a #pragma locale() statement. The locale is specified as a parameter of the form ll_cc where ll is a language code as defined in ISO 639-1:2002 , ISO649-2:1999 , or ISO 639-3:2007 and cc is a country code as defined in ISO 3166-1:2006 , ISO 3166-2:2007 , or ISO 3166-3:1999 .
#pragma locale()	Declares the locale used for a particular MOF file. The locale is specified as a parameter of the form ll_cc, where ll is a language code as defined in ISO 639-1:2002 , ISO649-2:1999 , or ISO 639-3:2007 and cc is a country code as defined in ISO 3166-1:2006 , ISO 3166-2:2007 , or ISO 3166-3:1999 . When the pragma is not specified, the assumed locale is "en_US". This pragma does not apply to the syntax structures of MOF. Keywords, such as "class" and "instance", are always in en_US.
#pragma namespace()	This pragma is used to specify a Namespace path.
#pragma nonlocal()	These compiler directives and the corresponding instance-level qualifiers were removed as an erratum in version 2.3.0 of this document.
#pragma nonlocaltype()	
#pragma source()	
#pragma sourcetype()	

4152 Pragma directives may be added as a MOF extension mechanism. Unless standardized in a future CIM
4153 infrastructure specification, such new pragma definitions must be considered vendor-specific. Use of non-
4154 standard pragma affects the interoperability of MOF import and export functions.

4155 7.11 Value Constants

4156 The constant types supported in the MOF syntax are described in the subclauses that follow. These are
4157 used in initializers for classes and instances and in the parameters to named qualifiers.

4158 For a formal specification of the representation, see ANNEX A.

4159 7.11.1 String Constants

4160 A string constant in MOF is represented as a sequence of one or more string constant parts, separated
4161 by whitespace or comments. Each string constant part is enclosed in double-quotes (") and contains zero

4162 or more UCS characters or escape sequences. Double quotes shall be escaped. The character repertoire
4163 for these UCS characters is defined in 5.2.2.

4164 The following escape sequences are defined for string constants:

```
4165      \b      // U+0008: backspace
4166      \t      // U+0009: horizontal tab
4167      \n      // U+000A: linefeed
4168      \f      // U+000C: form feed
4169      \r      // U+000D: carriage return
4170      \"      // U+0022: double quote (")
4171      \'      // U+0027: single quote (')
4172      \\      // U+005C: backslash (\)
4173      \x<hex> // a UCS character, where <hex> is one to four hex digits, representing its UCS code
4174              position
4175      \X<hex> // a UCS character, where <hex> is one to four hex digits, representing its UCS code
4176              position
```

4177 The `\x<hex>` and `\X<hex>` forms are limited to represent only the UCS-2 character set.

4178 For example, the following is a valid string constant:

```
4179      "This is a string"
```

4180 Successive quoted strings are concatenated as long as only whitespace or a comment intervenes:

```
4181      "This" " becomes a long string"
4182      "This" /* comment */ " becomes a long string"
```

4183 7.11.2 Character Constants

4184 A character constant in MOF is represented as one UCS character or escape sequence enclosed in
4185 single quotes ('), or as an integer constant as defined in 7.11.3. The character repertoire for the UCS
4186 character is defined in 5.2.3. The valid escape sequences are defined in 7.11.1. Single quotes shall be
4187 escaped. An integer constant represents the code position of a UCS character and its character
4188 repertoire is defined in 5.2.3.

4189 For example, the following are valid character constants:

```
4190      'a'      // U+0061: 'a'
4191      '\n'     // U+000A: linefeed
4192      '1'      // U+0031: '1'
4193      '\x32'   // U+0032: '2'
4194      65       // U+0041: 'A'
4195      0x41     // U+0041: 'A'
```

4196 7.11.3 Integer Constants

4197 Integer constants may be decimal, binary, octal, or hexadecimal. For example, the following constants are
4198 all legal:

```

4199     1000
4200     -12310
4201     0x100
4202     01236
4203     100101B

```

4204 Binary constants have a series of 1 and 0 digits, with a "b" or "B" suffix to indicate that the value is binary.

4205 The number of digits permitted depends on the current type of the expression. For example, it is not legal
4206 to assign the constant 0xFFFF to a property of type uint8.

4207 **7.11.4 Floating-Point Constants**

4208 Floating-point constants are declared as specified by [ANSI/IEEE 754-1985](#). For example, the following
4209 constants are legal:

```

4210     3.14
4211     -3.14
4212     -1.2778E+02

```

4213 The range for floating-point constants depends on whether float or double properties are used, and they
4214 must fit within the range specified for 4-byte and 8-byte floating-point values, respectively.

4215 **7.11.5 Object Reference Constants**

4216 As defined in 7.6.5, object references are special properties whose values are links or pointers to other
4217 objects, which may be classes or instances. Object reference constants are string representations of
4218 object paths for CIM MOF, as defined in 8.5.

4219 The usage of object reference constants as initializers for instance declarations is defined in 7.8, and as
4220 default values for properties in 7.5.3.

4221 **7.11.6 NULL**

4222 All types can be initialized to the predefined constant NULL, which indicates that no value is provided.
4223 The details of the internal implementation of the NULL value are not mandated by this document.

4224 **8 Naming**

4225 Because CIM is not bound to a particular technology or implementation, it promises to facilitate sharing
4226 management information among a variety of management platforms. The CIM naming mechanism
4227 addresses the following requirements:

- 4228 • Ability to unambiguously reference CIM objects residing in a CIM server.
- 4229 • Ability for CIM object names to be represented in multiple protocols, and for these
4230 representations the ability to be transformed across such protocols in an efficient manner.
- 4231 • Support the following types of CIM objects to be referenced: instances, classes, qualifier types
4232 and namespaces.
- 4233 • Ability to determine when two object names reference the same CIM object. This entails
4234 location transparency so that there is no need for a consumer of an object name to understand
4235 which management platforms proxy the instrumentation of other platforms.

4236 The Key qualifier is the CIM Meta-Model mechanism to identify the properties that uniquely identify an
4237 instance of a class (including an instance of an association) within a CIM namespace. This clause defines

4238 how CIM instances, classes, qualifier types and namespaces are referenced using the concept of CIM
4239 object paths.

4240 **8.1 CIM Namespaces**

4241 Because CIM allows multiple implementations, it is not sufficient to think of the name of a CIM instance as
4242 just the combination of its key properties. The instance name must also identify the implementation that
4243 actually hosts the instances. In order to separate the concept of a run-time container for CIM objects
4244 represented by a CIM server from the concept of naming, CIM defines the notion of a CIM namespace.
4245 This separation of concepts allows separating the design of a model along the boundaries of namespaces
4246 from the placement of namespaces in CIM servers.

4247 A namespace provides a scope of uniqueness for some types of object. Specifically, the names of class
4248 objects and of qualifier type objects shall be unique in a namespace. The compound key of instance
4249 objects shall be unique across all instances of the class (not including subclasses) within the namespace.

4250 In addition, a namespace is considered a CIM object since it is addressable using an object name.
4251 However, a namespace cannot host other namespaces, in other words the set of namespaces in a CIM
4252 server is flat. A namespace has a name which shall be unique within the CIM server.

4253 A namespace is also considered a run-time container within a CIM server which can host objects. For
4254 example, CIM objects are said to reside in namespaces as well as in CIM servers. Also, a common notion
4255 is to load the definition of qualifier types, classes and instances into a namespace, where they become
4256 objects that can be referenced. The run-time aspect of a CIM namespace makes it different from other
4257 definitions of namespace concepts that are addressing only the name uniqueness aspect, such as
4258 namespaces in Java, C++ or XML.

4259 **8.2 Naming CIM Objects**

4260 This subclause defines a concept for naming the objects residing in a CIM server. The naming concept
4261 allows for unambiguously referencing these objects and supports the following types of objects:

- 4262 • namespaces
- 4263 • qualifier types
- 4264 • classes
- 4265 • instances

4266 **8.2.1 Object Paths**

4267 The construct that references an object residing in a CIM server is called an object path. Since CIM is
4268 independent of implementations and protocols, object paths are defined in an abstract way that allows for
4269 defining different representations of the object paths. Protocols using object paths are expected to define
4270 representations of object paths as detailed in this subclause. A representation of object paths for CIM
4271 MOF is defined in 8.5.

4272 **DEPRECATED**

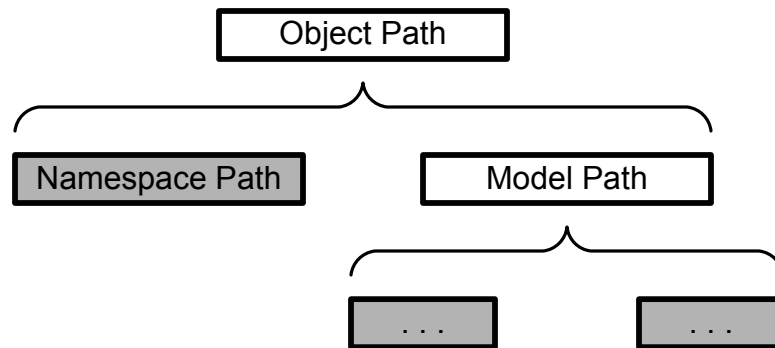
4273 Before version 2.6.0 of this document, object paths were referred to as "object names". The term "object
4274 name" is deprecated since version 2.6.0 of this document and the term "object path" should be used
4275 instead.

4276 **DEPRECATED**

4277 An object path is defined as a hierarchy of naming components. The leaf components in that hierarchy
4278 have a string value that is defined in this document. It is up to specifications using object paths to define

4279 how the string values of the leaf components are assembled into their own string representation of an
 4280 object path, as defined in 8.4.

4281 Figure 4 shows the general hierarchy of naming components of an object path. The naming components
 4282 are defined more specifically for each type of object supported by CIM naming. The leaf components are
 4283 shown with gray background.



4284

4285 **Figure 4 – General Component Structure of Object Path**

4286 Generally, an object path consists of two naming components:

- 4287 • namespace path – an unambiguous reference to the namespace in a CIM server, and
- 4288 • model path – an unambiguous identification of the object relative to that namespace.

4289 This document does not define the internal structure of a namespace path, but it defines requirements on
 4290 specifications using object paths in 8.4, including a requirement for a string representation of the
 4291 namespace path.

4292 A model path can be described using CIM model elements only. Therefore, this document defines the
 4293 naming components of the model path for each type of object supported by CIM naming. Since the leaf
 4294 components of model paths are CIM model elements, their string representation is well defined and
 4295 specifications using object paths only need to define how these strings are assembled into an object path,
 4296 as defined in 8.4.

4297 The definition of a string representation for object paths is left to specifications using object paths, as
 4298 described in 8.4.

4299 Two object paths match if their namespace path components match, and their model path components (if
 4300 any) have matching leaf components. As a result, two object paths that match reference the same CIM
 4301 object.

4302 NOTE: The matching of object paths is not just a simple string comparison of the string representations of object
 4303 paths.

4304 **8.2.2 Object Path for Namespace Objects**

4305 The object path for namespace objects is called namespace path. It consists of only the Namespace Path
 4306 component, as shown in Figure 5. A Model Path component is not present.



4307

4308 **Figure 5 – Component Structure of Object Path for Namespaces**

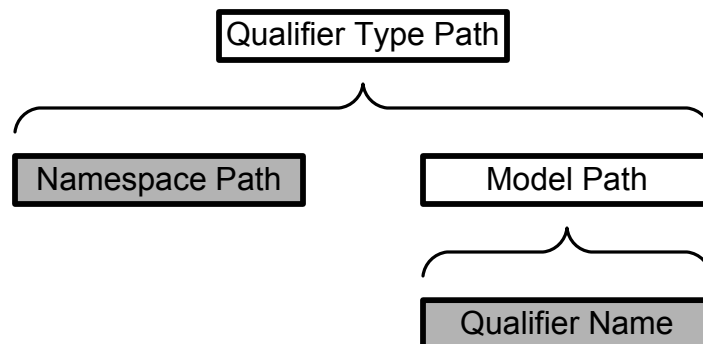
4309 The definition of a string representation for namespace paths is left to specifications using object paths,
4310 as described in 8.4.

4311 Two namespace paths match if they reference the same namespace. The definition of a method for
4312 determining whether two namespace paths reference the same namespace is left to specifications using
4313 object paths, as described in 8.4.

4314 The resulting method may or may not be able to determine whether two namespace paths reference the
4315 same namespace. For example, there may be alias names for namespaces, or different ports exposing
4316 access to the same namespace. Often, specifications using object paths need to revert to the minimally
4317 possible conclusion which is that namespace paths with equal string representations reference the same
4318 namespace, and that for namespace paths with unequal string representations no conclusion can be
4319 made about whether or not they reference the same namespace.

4320 **8.2.3 Object Path for Qualifier Type Objects**

4321 The object path for qualifier type objects is called qualifier type path. Its naming components have the
4322 structure defined in Figure 6.



4323

4324 **Figure 6 – Component Structure of Object Path for Qualifier Types**

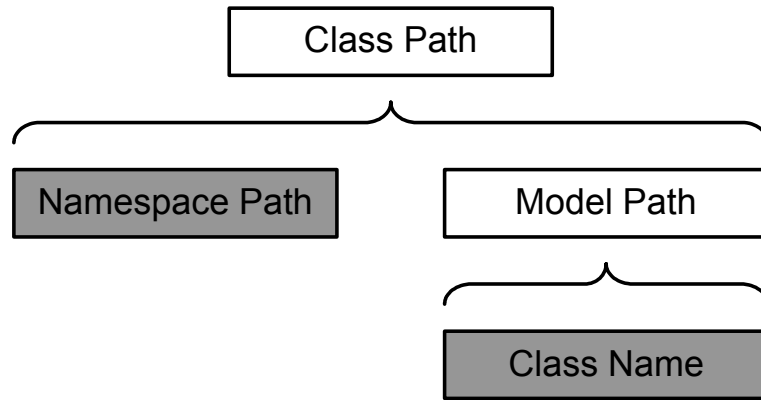
4325 The Namespace Path component is defined in 8.2.2.

4326 The string representation of the Qualifier Name component shall be the name of the qualifier, preserving
4327 the case defined in the namespace. For example, the string representation of the Qualifier Name
4328 component for the MappingStrings qualifier is "MappingStrings".

4329 Two Qualifier Names match as described in 8.2.6.

4330 8.2.4 Object Path for Class Objects

4331 The object path for class objects is called class path. Its naming components have the structure defined
4332 in Figure 7.



4333

4334 **Figure 7 – Component Structure of Object Path for Classes**

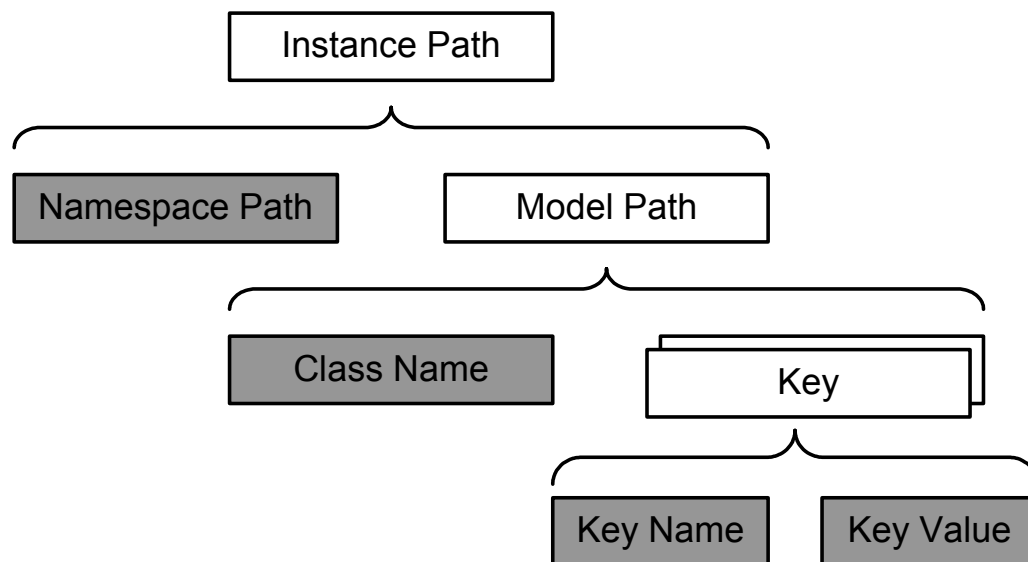
4335 The Namespace Path component is defined in 8.2.2.

4336 The string representation of the Qualifier Name component shall be the name of the qualifier, preserving
4337 the case defined in the namespace. For example, the string representation of the Qualifier Name
4338 component for the MappingStrings qualifier is "MappingStrings".

4339 Two Qualifier Names match as described in 8.2.6.

4340 8.2.5 Object Path for Class Objects

4341 The object path for class objects is called **class path**. Its naming components have the structure defined
4342 in Figure 7.



4343

4344

Figure 8 – Component Structure of Object Path for Instances

4345 The Namespace Path component is defined in 8.2.2.

4346 The Class Name component is defined in 8.2.4.

4347 The Model Path component consists of a Class Name component and an unordered set of one or more
 4348 Key components. There shall be one Key component for each key property (including references)
 4349 exposed by the class of the instance. The set of key properties includes any propagated keys, as defined
 4350 in 7.6.4. There shall not be Key components for properties (including references) that are not keys.
 4351 Classes that do not expose any keys cannot have instances that are addressable with an object path for
 4352 instances.

4353 The string representation of the Key Name component shall be the name of the key property, preserving
 4354 the case defined in the class residing in the namespace. For example, the string representation of the
 4355 Key Name component for a property ActualSpeed defined in a class ACME_Device is "ActualSpeed".

4356 Two Key Names match as described in 8.2.6.

4357 The Key Value component represents the value of the key property. The string representation of the Key
 4358 Value component is defined by specifications using object names, as defined in 8.4.

4359 Two Key Values match as defined for the datatype of the key property.

4360 **8.2.6 Matching CIM Names**

4361 Matching of CIM names (which consist of UCS characters) as defined in this document shall be
 4362 performed as if the following algorithm was applied:

4363 Any lower case UCS characters in the CIM names are translated to upper case.

4364 The CIM names are considered to match if the string identity matching rules defined in chapter 4 "String
 4365 Identity Matching" of [Character Model for the World Wide Web 1.0: Normalization](#) match when applied to
 4366 the upper case CIM names.

4367 In order to eliminate the costly processing involved in this, specifications using object paths may define
4368 simplified processing for applying this algorithm. One way to achieve this is to mandate that Normalization
4369 Form C (NFC), defined in [The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization](#)
4370 [Forms](#), which allows the normalization to be skipped when comparing the names.

4371 **8.3 Identity of CIM Objects**

4372 As defined in 8.2.1, two CIM objects are identical if their object paths match. Since this depends on
4373 whether their namespace paths match, it may not be possible to determine this (for details, see 8.2.2).

4374 Two different CIM objects (e.g., instances) can still represent aspects of the same managed object. In
4375 other words, identity at the level of CIM objects is separate from identity at the level of the represented
4376 managed objects.

4377 **8.4 Requirements on Specifications Using Object Paths**

4378 This subclause comprehensively defines the CIM naming related requirements on specifications using
4379 CIM object paths:

4380 Such specifications shall define a string representation of a namespace path (referred to as
4381 "namespace path string") using an ABNF syntax that defines its specification dependent
4382 components. The ABNF syntax shall not have any ABNF rules that are considered opaque or
4383 undefined. The ABNF syntax shall contain an ABNF rule for the namespace name.

4384 A namespace path string as defined with that ABNF syntax shall be able to reference a namespace
4385 object in a way that is unambiguous in the environment where the CIM server hosting the namespace is
4386 expected to be used. This typically translates to enterprise wide addressing using Internet Protocol
4387 addresses.

4388 Such specifications shall define a method for determining from the namespace path string the particular
4389 object path representation defined by the specification. This method should be based on the ABNF syntax
4390 defined for the namespace path string.

4391 Such specifications shall define a method for determining whether two namespace path strings reference
4392 the same namespace. As described in 8.2.2, this method may not support this in any case.

4393 Such specifications shall define how a string representation of the object paths for qualifier types, classes
4394 and instances is assembled from the string representations of the leaf components defined in 8.2.1 to
4395 8.2.5, using an ABNF syntax.

4396 Such specifications shall define string representations for all CIM datatypes that can be used as keys,
4397 using an ABNF syntax.

4398 **8.5 Object Paths Used in CIM MOF**

4399 Object paths are used in CIM MOF to reference instance objects in the following situations:

- 4400 • when specifying default values for references in association classes, as defined in 7.5.3.
- 4401 • when specifying initial values for references in association instances, as defined in 7.8.

4402 In CIM MOF, object paths are not used to reference namespace objects, class objects or qualifier type
4403 objects.

4404 The string representation of instance paths used in CIM MOF shall conform to the `WBEM-URI-`
4405 `UntypedInstancePath` ABNF rule defined in subclause 4.5 "Collected BNF for WBEM URI" of
4406 [DSP0207](#).

4407 That subclause also defines:

- 4408 • a string representation for the namespace path.
- 4409 • how a string representation of an instance path is assembled from the string representations of
4410 the leaf components defined in 8.2.1 to 8.2.5.
- 4411 • how the namespace name is determined from the string representation of an instance path.

4412 That specification does not presently define a method for determining whether two namespace path
4413 strings reference the same namespace.

4414 The string representations for key values shall be:

- 4415 • For the string datatype, as defined by the `stringValue` ABNF rule defined in ANNEX A, as
4416 one single string.
- 4417 • For the char16 datatype, as defined by the `charValue` ABNF rule defined in ANNEX A.
- 4418 • For the datetime datatype, the (unescaped) value of the datetime string as defined in 5.2.4, as
4419 one single string.
- 4420 • For the boolean datatype, as defined by the `booleanValue` ABNF rule defined in ANNEX A.
- 4421 • For integer datatypes, as defined by the `integerValue` ABNF rule defined in ANNEX A.
- 4422 • For real datatypes, as defined by the `realValue` ABNF rule defined in ANNEX A.
- 4423 • For <classname> REF datatypes, the string representation of the instance path as described in
4424 this subclause.

4425 EXAMPLE: Examples for string representations of instance paths in CIM MOF are as follows:

```
4426 "http://myserver.acme.com/root/cimv2:ACME_LogicalDisk.SystemName=\"acme\",Drive=\"C\""  
4427 "//myserver.acme.com:5988/root/cimv2:ACME_BooleanKeyClass.KeyProp=True"  
4428 "/root/cimv2:ACME_IntegerKeyClass.KeyProp=0x2A"  
4429 "ACME_CharKeyClass.KeyProp='\x41'"
```

4430 Instance paths referencing instances of association classes that have key references require special care
4431 regarding the escaping of the key values, which in this case are instance paths themselves. As defined in
4432 ANNEX A, the `objectHandle` ABNF rule is a string constant whose value conforms to the `objectName`
4433 ABNF rule. As defined in 7.11.1, representing a string value as a string in CIM MOF includes the
4434 escaping of any double quotes and backslashes present in the string value.

4435 EXAMPLE: The following example shows the string representation of an instance path referencing an instance of an
4436 association class with two key references. For better readability, the string is represented in three parts:

```
4437 "/root/cimv2:ACME_SystemDevice."  
4438 "System=\"/root/cimv2:ACME_System.Name=\\\\"acme\\\\""  
4439 ",Device=\"/root/cimv2:ACME_LogicalDisk.SystemName=\\\\"acme\\\\"\",Drive=\\\\"C\\\\"\""
```

4440 8.6 Mapping CIM Naming and Native Naming

4441 A managed environment may identify its managed objects in some way that is not necessarily the way
4442 they are identified in their CIM modeled appearance. The identification for managed objects used by the
4443 managed environment is called "native naming" in this document.

4444 At the level of interactions between a CIM client and a CIM server, CIM naming is used. This implies that
4445 a CIM server needs to be able to map CIM naming to the native naming used by the managed
4446 environment. This mapping needs to be performed in both directions: If a CIM operation references an
4447 instance with a CIM name, the CIM server needs to map the CIM name into the native name in order to
4448 reference the managed object by its native name. Similarly, if a CIM operation requests the enumeration

4449 of all instances of a class, the CIM server needs to map the native names by which the managed
4450 environment refers to the managed objects, into their CIM names before returning the enumerated
4451 instances.

4452 This subclause describes some techniques that can be used by CIM servers to map between CIM names
4453 and native names.

4454 **8.6.1 Native Name Contained in Opaque CIM Key**

4455 For CIM classes that have a single opaque key (e.g., InstanceId), it is possible to represent the native
4456 name in the opaque key in some (possibly class specific) way. This allows a CIM server to construct the
4457 native name from the key value, and vice versa.

4458 **8.6.2 Native Storage of CIM Name**

4459 If the native environment is able to maintain additional properties on its managed objects, the CIM name
4460 may be stored on each managed object as an additional property. For larger amounts of instances, this
4461 technique requires that there are lookup services available for the CIM server to look up managed objects
4462 by CIM name.

4463 **8.6.3 Translation Table**

4464 The CIM server can maintain a translation table between native names and CIM names, which allows to
4465 look up the names in both directions. Any entries created in the table are based on a defined mapping
4466 between native names and CIM names for the class. The entries in the table are automatically adjusted to
4467 the existence of instances as known by the CIM server.

4468 **8.6.4 No Mapping**

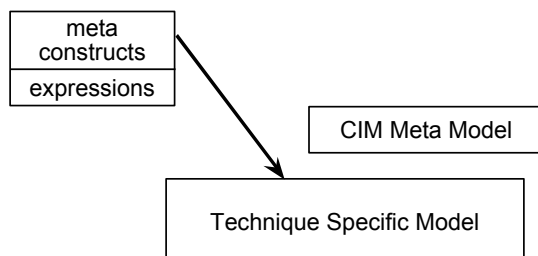
4469 Obviously, if the native naming is the same as the CIM naming, then no mapping needs to be performed.
4470 This may be the case for environments in which the native representation can be influenced to use CIM
4471 naming. An example for that is a relational database, where the relational model is defined such that CIM
4472 classes are used as tables, CIM properties as columns, and the index is defined on the columns
4473 corresponding to the key properties of the class.

4474 **9 Mapping Existing Models into CIM**

4475 Existing models have their own meta model and model. Three types of mappings can occur between
4476 meta schemas: technique, recast, and domain. Each mapping can be applied when MIF syntax is
4477 converted to MOF syntax.

4478 **9.1 Technique Mapping**

4479 A technique mapping uses the CIM meta-model constructs to describe the meta constructs of the source
4480 modeling technique (for example, MIF, GDMO, and SMI). Essentially, the CIM meta model is a meta
4481 meta-model for the source technique (see Figure 9).



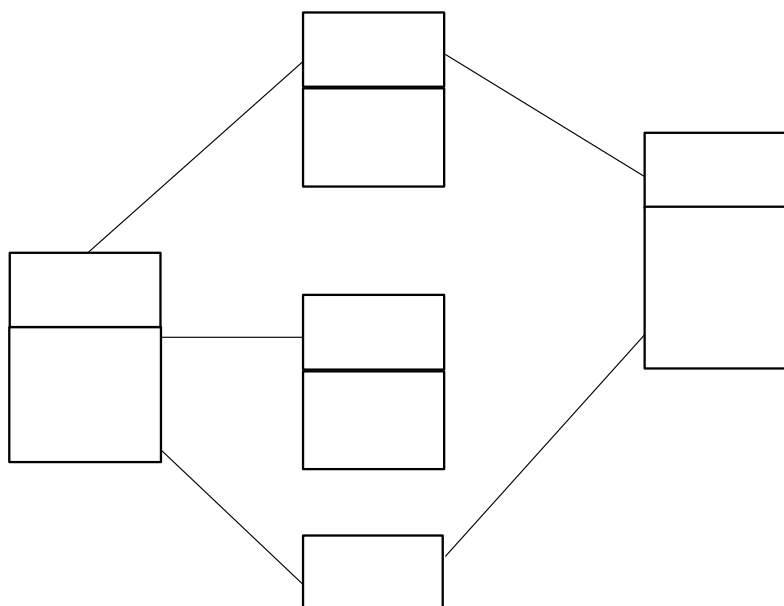
4482

4483

Figure 9 – Technique Mapping Example

4484 The DMTF uses the management information format (MIF) as the meta model to describe distributed
 4485 management information in a common way. Therefore, it is meaningful to describe a technique mapping
 4486 in which the CIM meta model is used to describe the MIF syntax.

4487 The mapping presented here takes the important types that can appear in a MIF file and then creates
 4488 classes for them. Thus, component, group, attribute, table, and enum are expressed in the CIM meta
 4489 model as classes. In addition, associations are defined to document how these classes are combined.
 4490 Figure 10 illustrates the results.



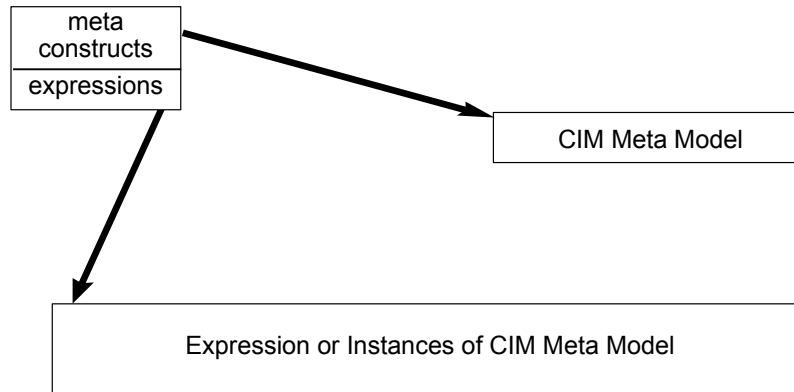
4491

4492

Figure 10 – MIF Technique Mapping Example

4493 9.2 Recast Mapping

4494 A recast mapping maps the meta constructs of the sources into the targeted meta constructs so that a
 4495 model expressed in the source can be translated into the target (Figure 11). The major design work is to
 4496 develop a mapping between the meta model of the sources and the CIM meta model. When this is done,
 4497 the source expressions are recast.



4498

4499

Figure 11 – Recast Mapping

4500 Following is an example of a recast mapping for MIF, assuming the following mapping:

4501 DMI attributes -> CIM properties
 4502 DMI key attributes -> CIM key properties
 4503 DMI groups -> CIM classes
 4504 DMI components -> CIM classes

4505 The standard DMI ComponentID group can be recast into a corresponding CIM class:

```

4506 Start Group
4507 Name = "ComponentID"
4508 Class = "DMTF|ComponentID|001"
4509 ID = 1
4510 Description = "This group defines the attributes common to all "
4511 "components. This group is required."
4512 Start Attribute
4513 Name = "Manufacturer"
4514 ID = 1
4515 Description = "Manufacturer of this system."
4516 Access = Read-Only
4517 Storage = Common
4518 Type = DisplayString(64)
4519 Value = ""
4520 End Attribute
4521 Start Attribute
4522 Name = "Product"
4523 ID = 2
4524 Description = "Product name for this system."
4525 Access = Read-Only
4526 Storage = Common
4527 Type = DisplayString(64)
4528 Value = ""
4529 End Attribute
4530 Start Attribute
4531 Name = "Version"
4532 ID = 3
4533 Description = "Version number of this system."
4534 Access = Read-Only
  
```

```

4535     Storage = Specific
4536     Type = DisplayString(64)
4537     Value = ""
4538 End Attribute
4539 Start Attribute
4540     Name = "Serial Number"
4541     ID = 4
4542     Description = "Serial number for this system."
4543     Access = Read-Only
4544     Storage = Specific
4545     Type = DisplayString(64)
4546     Value = ""
4547 End Attribute
4548 Start Attribute
4549     Name = "Installation"
4550     ID = 5
4551     Description = "Component installation time and date."
4552     Access = Read-Only
4553     Storage = Specific
4554     Type = Date
4555     Value = ""
4556 End Attribute
4557 Start Attribute
4558     Name = "Verify"
4559     ID = 6
4560     Description = "A code that provides a level of verification that the "
4561         "component is still installed and working."
4562     Access = Read-Only
4563     Storage = Common
4564     Type = Start ENUM
4565         0 = "An error occurred; check status code."
4566         1 = "This component does not exist."
4567         2 = "Verification is not supported."
4568         3 = "Reserved."
4569         4 = "This component exists, but the functionality is untested."
4570         5 = "This component exists, but the functionality is unknown."
4571         6 = "This component exists, and is not functioning correctly."
4572         7 = "This component exists, and is functioning correctly."
4573     End ENUM
4574     Value = 1
4575 End Attribute
4576 End Group

```

4577 A corresponding CIM class might be the following. Notice that properties in the example include an ID
4578 qualifier to represent the ID of the corresponding DMI attribute. Here, a user-defined qualifier may be
4579 necessary:

```

4580 [Name ("ComponentID"), ID (1), Description (
4581     "This group defines the attributes common to all components. "
4582     "This group is required.")]
4583 class DMTF|ComponentID|001 {
4584     [ID (1), Description ("Manufacturer of this system."), maxlen (64)]
4585     string Manufacturer;
4586     [ID (2), Description ("Product name for this system."), maxlen (64)]
4587     string Product;
4588     [ID (3), Description ("Version number of this system."), maxlen (64)]

```

```

4589     string Version;
4590     [ID (4), Description ("Serial number for this system."), maxlen (64)]
4591     string Serial_Number;
4592     [ID (5), Description("Component installation time and date.")]
4593     datetime Installation;
4594     [ID (6), Description("A code that provides a level of verification "
4595     "that the component is still installed and working."),
4596     Value (1)]
4597     string Verify;
4598 };
    
```

4599 **9.3 Domain Mapping**

4600 A domain mapping takes a source expressed in a particular technique and maps its content into either the
 4601 core or common models or extension sub-schemas of the CIM. This mapping does not rely heavily on a
 4602 meta-to-meta mapping; it is primarily a content-to-content mapping. In one case, the mapping is actually a
 4603 re-expression of content in a more common way using a more expressive technique.

4604 Following is an example of how DMI can supply CIM properties using information from the DMI disks
 4605 group ("DMTF|Disks|002"). For a hypothetical CIM disk class, the CIM properties are expressed as shown
 4606 in Table 11.

4607 **Table 11 – 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"

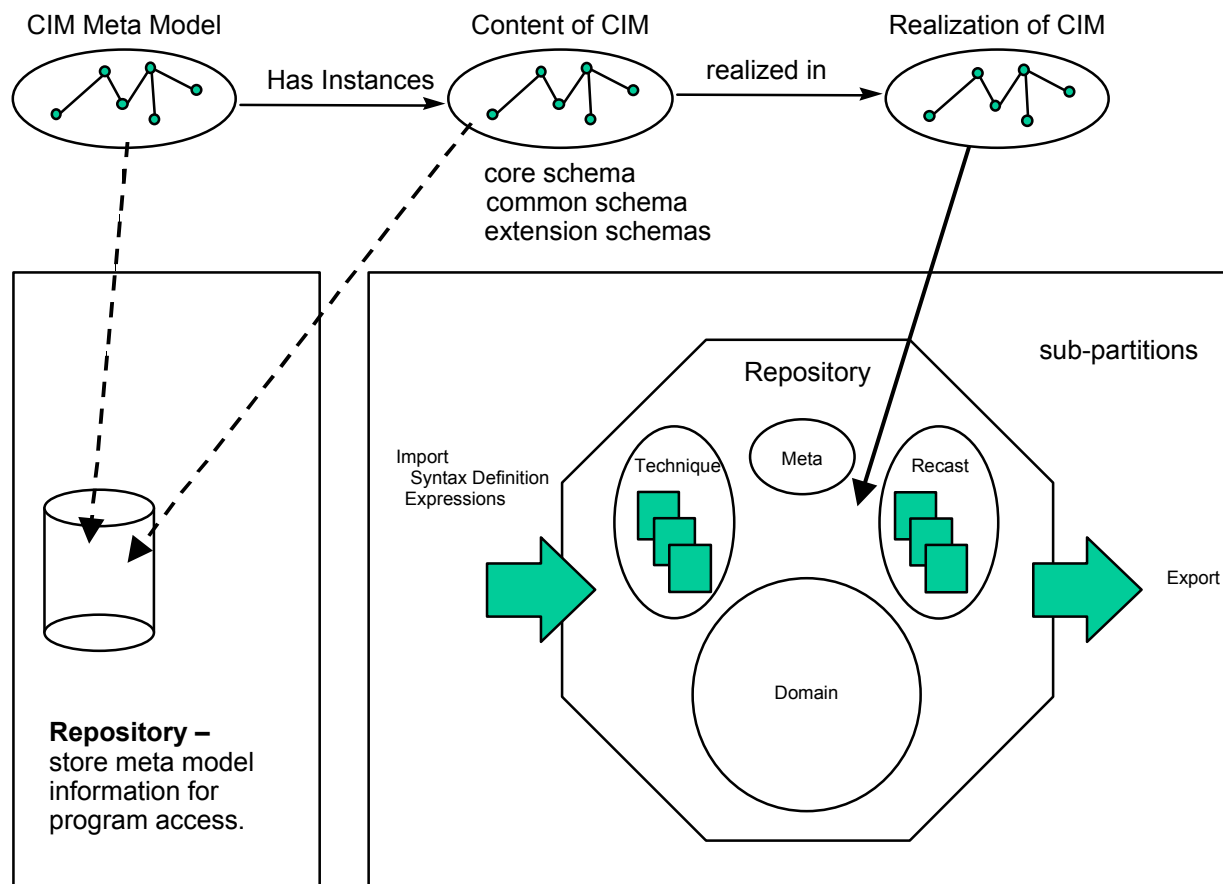
4608 **9.4 Mapping Scratch Pads**

4609 In general, when the contents of models are mapped between different meta schemas, information is lost
 4610 or missing. To fill this gap, scratch pads are expressed in the CIM meta model using qualifiers, which are
 4611 actually extensions to the meta model (for example, see 10.2). These scratch pads are critical to the
 4612 exchange of core, common, and extension model content with the various technologies used to build
 4613 management applications.

4614 **10 Repository Perspective**

4615 This clause describes a repository and presents a complete picture of the potential to exploit it. A
 4616 repository stores definitions and structural information, and it includes the capability to extract the
 4617 definitions in a form that is useful to application developers. Some repositories allow the definitions to be
 4618 imported into and exported from the repository in multiple forms. The notions of importing and exporting
 4619 can be refined so that they distinguish between three types of mappings.

4620 Using the mapping definitions in Clause 9, the repository can be organized into the four partitions: meta,
 4621 technique, recast, and domain (see Figure 12).



4622

4623

Figure 12 – Repository Partitions

4624 The repository partitions have the following characteristics:

4625 • Each partition is discrete:

4626 – The meta partition refers to the definitions of the CIM meta model.

4627 – The technique partition refers to definitions that are loaded using technique mappings.

4628 – The recast partition refers to definitions that are loaded using recast mappings.

4629 – The domain partition refers to the definitions associated with the core and common models
4630 and the extension schemas.

4631 • The technique and recast partitions can be organized into multiple sub-partitions to capture
4632 each source uniquely. For example, there is a technique sub-partition for each unique meta
4633 language encountered (that is, one for MIF, one for GDMO, one for SMI, and so on). In the re-
4634 cast partition, there is a sub-partition for each meta language.

4635 • The act of importing the content of an existing source can result in entries in the recast or
4636 domain partition.

4637 10.1 DMTF MIF Mapping Strategies

4638 When the meta-model definition and the baseline for the CIM schema are complete, the next step is to
4639 map another source of management information (such as standard groups) into the repository. The main
4640 goal is to do the work required to import one or more of the standard groups. The possible import
4641 scenarios for a DMTF standard group are as follows:

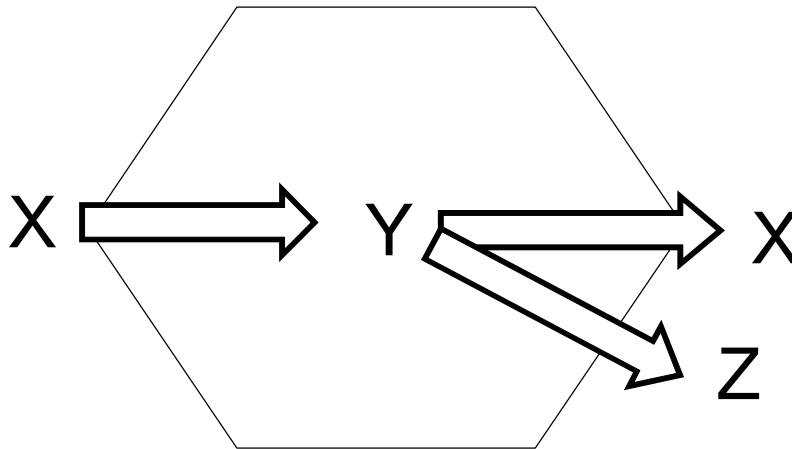
- 4642 • *To Technique Partition:* Create a technique mapping for the MIF syntax that is the same for all
4643 standard groups and needs to be updated only if the MIF syntax changes.
- 4644 • *To Recast Partition:* Create a recast mapping from a particular standard group into a sub-
4645 partition of the recast partition. This mapping allows the entire contents of the selected group to
4646 be loaded into a sub-partition of the recast partition. The same algorithm can be used to map
4647 additional standard groups into that same sub-partition.
- 4648 • *To Domain Partition:* Create a domain mapping for the content of a particular standard group
4649 that overlaps with the content of the CIM schema.
- 4650 • *To Domain Partition:* Create a domain mapping for the content of a particular standard group
4651 that does not overlap with CIM schema into an extension sub-schema.
- 4652 • *To Domain Partition:* Propose extensions to the content of the CIM schema and then create a
4653 domain mapping.

4654 Any combination of these five scenarios can be initiated by a team that is responsible for mapping an
4655 existing source into the CIM repository. Many other details must be addressed as the content of any of
4656 the sources changes or when the core or common model changes. When numerous existing sources are
4657 imported using all the import scenarios, we must consider the export side. Ignoring the technique
4658 partition, the possible export scenarios are as follows:

- 4659 • *From Recast Partition:* Create a recast mapping for a sub-partition in the recast partition to a
4660 standard group (that is, inverse of import 2). The desired method is to use the recast mapping to
4661 translate a standard group into a GDMO definition.
- 4662 • *From Recast Partition:* Create a domain mapping for a recast sub-partition to a known
4663 management model that is not the original source for the content that overlaps.
- 4664 • *From Domain Partition:* Create a recast mapping for the complete contents of the CIM schema
4665 to a selected technique (for MIF, this remapping results in a non-standard group).
- 4666 • *From Domain Partition:* Create a domain mapping for the contents of the CIM schema that
4667 overlaps with the content of an existing management model.
- 4668 • *From Domain Partition:* Create a domain mapping for the entire contents of the CIM schema to
4669 an existing management model with the necessary extensions.

4670 10.2 Recording Mapping Decisions

4671 To understand the role of the scratch pad in the repository (see Figure 13), it is necessary to look at the
4672 import and export scenarios for the different partitions in the repository (technique, recast, and
4673 application). These mappings can be organized into two categories: homogeneous and heterogeneous.
4674 In the homogeneous category, the imported syntax and expressions are the same as the exported syntax
4675 and expressions (for example, software MIF in and software MIF out). In the heterogeneous category, the
4676 imported syntax and expressions are different from the exported syntax and expressions (for example,
4677 MIF in and GDMO out). For the homogenous category, the information can be recorded by creating
4678 qualifiers during an import operation so the content can be exported properly. For the heterogeneous
4679 category, the qualifiers must be added after the content is loaded into a partition of the repository.
4680 Figure 13 shows the X schema imported into the Y schema and then homogeneously exported into X or
4681 heterogeneously exported into Z. Each export arrow works with a different scratch pad.

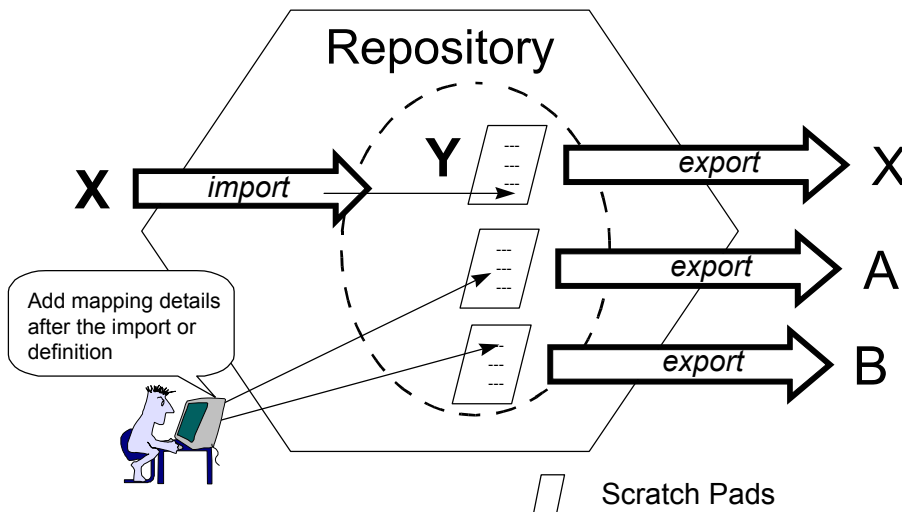


4682

4683

Figure 13 – Homogeneous and Heterogeneous Export

4684 The definition of the heterogeneous category is actually based on knowing how a schema is loaded into
 4685 the repository. To assist in understanding the export process, we can think of this process as using one of
 4686 multiple scratch pads. One scratch pad is created when the schema is loaded, and the others are added
 4687 to handle mappings to schema techniques other than the import source (Figure 14).



import

4688

4689

Figure 14 – Scratch Pads and Mapping

4690 Figure 14 shows how the scratch pads of qualifiers are used without factoring in the unique aspects of
 4691 each partition (technique, recast, applications) within the CIM repository. The next step is to consider
 4692 these partitions.

4693 For the technique partition, there is no need for a scratch pad because the CIM meta model is used to
 4694 describe the constructs in the source meta schema. Therefore, by definition, there is one homogeneous
 4695 mapping for each meta schema covered by the technique partition. These mappings create CIM objects

4696 for the syntactic constructs of the schema and create associations for the ways they can be combined.
4697 (For example, MIF groups include attributes.)

4698 For the recast partition, there are multiple scratch pads for each sub-partition because one is required for
4699 each export target and there can be multiple mapping algorithms for each target. Multiple mapping
4700 algorithms occur because part of creating a recast mapping involves mapping the constructs of the
4701 source into CIM meta-model constructs. Therefore, for the MIF syntax, a mapping must be created for
4702 component, group, attribute, and so on, into appropriate CIM meta-model constructs such as object,
4703 association, property, and so on. These mappings can be arbitrary. For example, one decision to be
4704 made is whether a group or a component maps into an object. Two different recast mapping algorithms
4705 are possible: one that maps groups into objects with qualifiers that preserve the component, and one that
4706 maps components into objects with qualifiers that preserve the group name for the properties. Therefore,
4707 the scratch pads in the recast partition are organized by target technique and employed algorithm.

4708 For the domain partitions, there are two types of mappings:

- 4709 • A mapping similar to the recast partition in that part of the domain partition is mapped into the
4710 syntax of another meta schema. These mappings can use the same qualifier scratch pads and
4711 associated algorithms that are developed for the recast partition.
- 4712 • A mapping that facilitates documenting the content overlap between the domain partition and
4713 another model (for example, software groups).

4714 These mappings cannot be determined in a generic way at import time; therefore, it is best to consider
4715 them in the context of exporting. The mapping uses filters to determine the overlaps and then performs
4716 the necessary conversions. The filtering can use qualifiers to indicate that a particular set of domain
4717 partition constructs maps into a combination of constructs in the target/source model. The conversions
4718 are documented in the repository using a complex set of qualifiers that capture how to write or insert the
4719 overlapped content into the target model. The mapping qualifiers for the domain partition are organized
4720 like the recasting partition for the syntax conversions, and there is a scratch pad for each model for
4721 documenting overlapping content.

4722 In summary, pick the partition, develop a mapping, and identify the qualifiers necessary to capture
4723 potentially lost information when mapping details are developed for a particular source. On the export
4724 side, the mapping algorithm verifies whether the content to be exported includes the necessary qualifiers
4725 for the logic to work.

4726

ANNEX A (normative)

MOF Syntax Grammar Description

4727
4728
4729
4730

4731 This annex presents the grammar for MOF syntax. While the grammar is convenient for describing the
4732 MOF syntax clearly, the same MOF language can also be described by a different, LL(1)-parsable,
4733 grammar, which enables low-footprint implementations of MOF compilers. In addition, the following
4734 applies:

- 4735 1) All keywords are case-insensitive.
- 4736 2) In the current release, the MOF syntax does not support initializing an array value to empty (an
4737 array with no elements). In version 3 of this document, the DMTF plans to extend the MOF
4738 syntax to support this functionality as follows:

4739 arrayInitialize = "{" [arrayElementList] "}"

4740 arrayElementList = constantValue *("," constantValue)

4741 To ensure interoperability with implementations of version 2 of this document, the DMTF
4742 recommends that, where possible, the value of NULL rather than empty ({}) be used to
4743 represent the most common use cases. However, if this practice should cause confusion or
4744 other issues, implementations may use the syntax of version 3 of this document to initialize an
4745 empty array.

4746 The following is the grammar for the MOF syntax, defined in ABNF. Unless otherwise stated, the ABNF in
4747 this annex has whitespace allowed.

4748

```

mofSpecification      = *mofProduction

mofProduction        = compilerDirective /
                       classDeclaration /
                       assocDeclaration /
                       indicDeclaration /
                       qualifierDeclaration /
                       instanceDeclaration

compilerDirective     = PRAGMA pragmaName "(" pragmaParameter ")"

pragmaName            = IDENTIFIER

pragmaParameter       = stringValue

classDeclaration      = [ qualifierList ]
                       CLASS className [ superClass ]
                       "{" *classFeature "}" ";"

assocDeclaration      = "[" ASSOCIATION *( "," qualifier ) "]"
                       CLASS className [ superClass ]
                       "{" *associationFeature "}" ";"
                       ; Context:

```

```

; The remaining qualifier list must not include
; the ASSOCIATION qualifier again. If the
; association has no super association, then at
; least two references must be specified! The
; ASSOCIATION qualifier may be omitted in
; sub-associations.

indicDeclaration      = "[" INDICATION *( "," qualifier ) "]"
                      CLASS className [ superClass ]
                      "{" *classFeature "}" ";"

namespaceName        = IDENTIFIER *( "/" IDENTIFIER )

className             = schemaName "_" IDENTIFIER ; NO whitespace !
                      ; Context:
                      ; Schema name must not include "_" !

alias                 = AS aliasIdentifier

aliasIdentifier       = "$" IDENTIFIER ; NO whitespace !

superClass            = ":" className

classFeature           = propertyDeclaration / methodDeclaration

associationFeature    = classFeature / referenceDeclaration

qualifierList         = "[" qualifier *( "," qualifier ) "]"

qualifier              = qualifierName [ qualifierParameter ] [ ":" 1*flavor ]
                      ; DEPRECATED: The ABNF rule [ ":" 1*flavor ] is used
                      ; for the concept of implicitly defined qualifier types
                      ; and is deprecated. See 5.1.2.16 for details.

qualifierParameter    = "(" constantValue ")" / arrayInitializer

flavor                = ENABLEOVERRIDE / DISABLEOVERRIDE / RESTRICTED /
                      TOSUBCLASS / TRANSLATABLE

propertyDeclaration   = [ qualifierList ] dataType propertyName
                      [ array ] [ defaultValue ] ";"

referenceDeclaration  = [ qualifierList ] objectRef referenceName
                      [ defaultValue ] ";"

methodDeclaration     = [ qualifierList ] dataType methodName
                      "(" [ parameterList ] ")" ";"

propertyName         = IDENTIFIER

referenceName         = IDENTIFIER

methodName            = IDENTIFIER

dataType              = DT_UINT8 / DT_SINT8 / DT_UINT16 / DT_SINT16 /
                      DT_UINT32 / DT_SINT32 / DT_UINT64 / DT_SINT64 /
                      DT_REAL32 / DT_REAL64 / DT_CHAR16 /
                      DT_STR / DT_BOOL / DT_DATETIME

objectRef             = className REF

parameterList         = parameter *( "," parameter )

```

```

parameter          = [ qualifierList ] ( dataType / objectRef ) parameterName
                    [ array ]

parameterName      = IDENTIFIER

array              = "[" [positiveDecimalValue] "]"

positiveDecimalValue = positiveDecimalDigit *decimalDigit

defaultValue       = "=" initializer

initializer        = ConstantValue / arrayInitializer / referenceInitializer

arrayInitializer   = "{" constantValue*( "," constantValue)"}"

constantValue      = integerValue / realValue / charValue / stringValue /
                    datetimeValue / booleanValue / nullValue

integerValue       = binaryValue / octalValue / decimalValue / hexValue

referenceInitializer = objectPath / aliasIdentifier

objectPath         = stringValue
                    ; the(unescaped)contents of stringValue shall conform
                    ; to the string representation for object paths as
                    ; defined in 8.5.

qualifierDeclaration = QUALIFIER qualifierName qualifierType scope
                    [ defaultFlavor ] ";"

qualifierName      = IDENTIFIER

qualifierType      = ":" dataType [ array ] [ defaultValue ]

scope              = "," SCOPE "(" metaElement *( "," metaElement ) ")"

metaElement        = CLASS / ASSOCIATION / INDICATION / QUALIFIER
                    PROPERTY / REFERENCE / METHOD / PARAMETER / ANY

defaultFlavor      = "," FLAVOR "(" flavor *( "," flavor ) ")"

instanceDeclaration = [ qualifierList ] INSTANCE OF className [ alias ]
                    "{" 1*valueInitializer }" ";"

valueInitializer   = [ qualifierList ]
                    ( propertyName / referenceName ) "=" initializer ";"

```

4749 These ABNF rules do not allow whitespace, unless stated otherwise:

4750

```

schemaName         = IDENTIFIER
                    ; Context:
                    ; Schema name must not include "_" !

fileName           = stringValue

binaryValue        = [ "+" / "-" ] 1*binaryDigit ( "b" / "B" )

binaryDigit        = "0" / "1"

octalValue         = [ "+" / "-" ] "0" 1*octalDigit

octalDigit         = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"

```

decimalValue	=	["+" / "-"] (positiveDecimalDigit *decimalDigit / "0")
decimalDigit	=	"0" / positiveDecimalDigit
positiveDecimalDigit	=	"1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"
hexValue	=	["+" / "-"] ("0x" / "0X") 1*hexDigit
hexDigit	=	decimalDigit / "a" / "A" / "b" / "B" / "c" / "C" / "d" / "D" / "e" / "E" / "f" / "F"
realValue	=	["+" / "-"] *decimalDigit "." 1*decimalDigit [("e" / "E") ["+" / "-"] 1*decimalDigit]
charValue	=	"'" char16Char "'" / integerValue ; Single quotes shall be escaped. ; For details, see 7.11.2
stringValue	=	1*("" *stringChar "") ; Whitespace and comment is allowed between double ; quoted parts. ; Double quotes shall be escaped. ; For details, see 7.11.1
stringChar	=	UCScharString / stringEscapeSequence
Char16Char	=	UCScharChar16 / stringEscapeSequence
UCScharString		is any UCS character for use in string constants as defined in 7.11.1.
UCScharChar16		is any UCS character for use in char16 constants as defined in 7.11.2.
stringEscapeSequence		is any escape sequence for string and char16 constants, as defined in 7.11.1.
booleanValue	=	TRUE / FALSE
nullValue	=	NULL
IDENTIFIER	=	firstIdentifierChar *(nextIdentifierChar)
firstIdentifierChar	=	UPPERALPHA / LOWERALPHA / UNDERSCORE / UCS0080TOFFEF ; DEPRECATED: The use of the UCS0080TOFFEF ABNF rule ; within the firstIdentifierChar ABNF rule is deprecated ; since version 2.6.0 of this document.
nextIdentifierChar	=	firstIdentifierChar / DIGIT
UPPERALPHA	=	U+0041...U+005A ; "A" ... "Z"
LOWERALPHA	=	U+0061...U+007A ; "a" ... "z"
UNDERSCORE	=	U+005F ; "_"
DIGIT	=	U+0030...U+0039 ; "0" ... "9"
UCS0080TOFFEF		is any assigned UCS character with code positions in the range U+0080..U+FFEF

```

datetimeValue      = 1*( "" *stringChar "" )
                    ; Whitespace is allowed between the double quoted parts.
                    ; The combined string value shall conform to the format
                    ; defined by the dt-format ABNF rule.

dt-format          = dt-timestampValue / dt-intervalValue

dt-timestampValue  = 14*14(decimalDigit) "." dt-microseconds
                    ("+" / "-") dt-timezone /
                    dt-yyyymmddhhmmss "." 6*6("") ("+" / "-") dt-timezone
                    ; With further constraints on the field values
                    ; as defined in subclause 5.2.4.

dt-intervalValue   = 14*14(decimalDigit) "." dt-microseconds ":" "000" /
                    dt-ddddddddhhmmss "." 6*6("") ":" "000"
                    ; With further constraints on the field values
                    ; as defined in subclause 5.2.4.

dt-yyyymmddhhmmss = 12*12(decimalDigit) 2*2("") /
                    10*10(decimalDigit) 4*4("") /
                    8*8(decimalDigit) 6*6("") /
                    6*6(decimalDigit) 8*8("") /
                    4*4(decimalDigit) 10*10("") /
                    14*14("")

dt-ddddddddhhmmss = 12*12(decimalDigit) 2*2("") /
                    10*10(decimalDigit) 4*4("") /
                    8*8(decimalDigit) 6*6("") /
                    14*14("")

dt-microseconds    = 6*6(decimalDigit) /
                    5*5(decimalDigit) 1*1("") /
                    4*4(decimalDigit) 2*2("") /
                    3*3(decimalDigit) 3*3("") /
                    2*2(decimalDigit) 4*4("") /
                    1*1(decimalDigit) 5*5("") /
                    6*6("")

dt-timezone        = 3*3(decimalDigit)

```

4751 The remaining ABNF rules are case-insensitive keywords:

```

ANY                = "any"
AS                 = "as"
ASSOCIATION        = "association"
CLASS              = "class"
DISABLEOVERRIDE    = "disableOverride"
DT_BOOL            = "boolean"
DT_CHAR16          = "char16"
DT_DATETIME        = "datetime"
DT_REAL32          = "real32"
DT_REAL64          = "real64"
DT_SINT16          = "sint16"
DT_SINT32          = "sint32"
DT_SINT64          = "sint64"

```

DT_SINT8	=	"sint8"
DT_STR	=	"string"
DT_UINT16	=	"uint16"
DT_UINT32	=	"uint32"
DT_UINT64	=	"uint64"
DT_UINT8	=	"uint8"
ENABLEOVERRIDE	=	"enableoverride"
FALSE	=	"false"
FLAVOR	=	"flavor"
INDICATION	=	"indication"
INSTANCE	=	"instance"
METHOD	=	"method"
NULL	=	"null"
OF	=	"of"
PARAMETER	=	"parameter"
PRAGMA	=	"#pragma"
PROPERTY	=	"property"
QUALIFIER	=	"qualifier"
REF	=	"ref"
REFERENCE	=	"reference"
RESTRICTED	=	"restricted"
SCHEMA	=	"schema"
SCOPE	=	"scope"
TOSUBCLASS	=	"tosubclass"
TRANSLATABLE	=	"translatable"
TRUE	=	"true"

ANNEX B (informative)

CIM Meta Schema

4752
4753
4754
4755

4756 This annex defines a CIM model that represents the CIM meta schema defined in 5.1. UML associations
4757 are represented as CIM associations.

4758 CIM associations always own their association ends (i.e., the CIM references), while in UML, they are
4759 owned either by the association or by the associated class. For sake of simplicity of the description, the
4760 UML definition of the CIM meta schema defined in 5.1 had the association ends owned by the associated
4761 classes. The CIM model defined in this annex has no other choice but having them owned by the
4762 associations. The resulting CIM model is still a correct description of the CIM meta schema.

```

4763     [Version("2.6.0"), Abstract, Description (
4764     "Abstract class for CIM elements, providing the ability for "
4765     "an element to have a name.\n"
4766     "Some kinds of elements provide the ability to have qualifiers "
4767     "specified on them, as described in subclasses of "
4768     "Meta_NamedElement.") ]
4769 class Meta_NamedElement
4770 {
4771     [Required, Description (
4772     "The name of the element. The format of the name is "
4773     "determined by subclasses of Meta_NamedElement.\n"
4774     "The names of elements shall be compared "
4775     "case-insensitively.") ]
4776     string Name;
4777 };
4778
4779 // =====
4780 //     TypedElement
4781 // =====
4782     [Version("2.6.0"), Abstract, Description (
4783     "Abstract class for CIM elements that have a CIM data "
4784     "type.\n"
4785     "Not all kinds of CIM data types may be used for all kinds of "
4786     "typed elements. The details are determined by subclasses of "
4787     "Meta_TypedElement.") ]
4788 class Meta_TypedElement : Meta_NamedElement
4789 {
4790 };
4791
4792 // =====
4793 //     Type
4794 // =====
4795     [Version("2.6.0"), Abstract, Description (
4796     "Abstract class for any CIM data types, including arrays of "
```

```

4797     "such."),
4798     ClassConstraint {
4799     /* If the type is no array type, the value of ArraySize shall "
4800     "be NULL. */\n"
4801     "inv: self.IsArray = false\n"
4802     "    implies self.ArraySize.IsNull()"} ]
4803     /* A Type instance shall be owned by only one owner. */\n"
4804     "inv: self.Meta_ElementType[OwnedType].OwningElement->size() +\n"
4805     "    self.Meta_ValueType[OwnedType].OwningValue->size() = 1"} ]
4806 class Meta_Type
4807 {
4808     [Required, Description (
4809     "Indicates whether the type is an array type. For details "
4810     "on arrays, see 7.8.2.") ]") ]
4811     boolean IsArray;
4812
4813     [Description (
4814     "If the type is an array type, a non-NULL value indicates "
4815     "the size of a fixed-size array, and a NULL value indicates "
4816     "a variable-length array. For details on arrays, see "
4817     "7.8.2.") ]
4818     sint64 ArraySize;
4819 };
4820
4821 // =====
4822 //     PrimitiveType
4823 // =====
4824 [Version("2.6.0"), Description (
4825 "A CIM data type that is one of the intrinsic types defined in "
4826 "Table 2, excluding references."),
4827 ClassConstraint {
4828 /* This kind of type shall be used only for the following "
4829 "kinds of typed elements: Method, Parameter, ordinary Property, "
4830 "and QualifierType. */\n"
4831 "inv: let e : Meta_NamedElement =\n"
4832 "    self.Meta_ElementType[OwnedType].OwningElement\n"
4833 "    in\n"
4834 "    e.oclIsTypeOf(Meta_Method) or\n"
4835 "    e.oclIsTypeOf(Meta_Parameter) or\n"
4836 "    e.oclIsTypeOf(Meta_Property) or\n"
4837 "    e.oclIsTypeOf(Meta_QualifierType)"} ]
4838 class Meta_PrimitiveType : Meta_Type
4839 {
4840     [Required, Description (
4841     "The name of the CIM data type.\n"
4842     "The type name shall follow the formal syntax defined by "
4843     "the dataType ABNF rule in ANNEX A.") ]
4844     string TypeName;
4845 };

```

```

4846
4847 // =====
4848 //   ReferenceType
4849 // =====
4850 [Version("2.6.0"), Description (
4851   "A CIM data type that is a reference, as defined in Table 2."),
4852   ClassConstraint {
4853     /* This kind of type shall be used only for the following "
4854     "kinds of typed elements: Parameter and Reference. */\n"
4855     "inv: let e : Meta_NamedElement = /* the typed element */\n"
4856     "   self.Meta_ElementType[OwnedType].OwningElement\n"
4857     "   in\n"
4858     "   e.oclIsTypeOf(Meta_Parameter) or\n"
4859     "   e.oclIsTypeOf(Meta_Reference)",
4860     /* When used for a Reference, the type shall not be an "
4861     "array. */\n"
4862     "inv: self.Meta_ElementType[OwnedType].OwningElement.\n"
4863     "   oclIsTypeOf(Meta_Reference)\n"
4864     "   implies\n"
4865     "   self.IsArray = false"} ]
4866 class Meta_ReferenceType : Meta_Type
4867 {
4868 };
4869 // =====
4870 //   Schema
4871 // =====
4872 [Version("2.6.0"), Description (
4873   "Models a CIM schema. A CIM schema is a set of CIM classes with "
4874   "a single defining authority or owning organization."),
4875   ClassConstraint {
4876     /* The elements owned by a schema shall be only of kind "
4877     "Class. */\n"
4878     "inv: self.Meta_SchemaElement[OwningSchema].OwnedElement.\n"
4879     "   oclIsTypeOf(Meta_Class)"} ]
4880 class Meta_Schema : Meta_NamedElement
4881 {
4882   [Override ("Name"), Description (
4883     "The name of the schema. The schema name shall follow the "
4884     "formal syntax defined by the schemaName ABNF rule in "
4885     "ANNEX A.\n"
4886     "Schema names shall be compared case insensitively.") ]
4887   string Name;
4888 };
4889 // =====
4890 //   Class
4891 // =====
4892
4893 [Version("2.6.0"), Description (

```

```

4895     "Models a CIM class. A CIM class is a common type for a set of "
4896     "CIM instances that support the same features (i.e. properties "
4897     "and methods). A CIM class models an aspect of a managed "
4898     "element.\n"
4899     "Classes may be arranged in a generalization hierarchy that "
4900     "represents subtype relationships between classes. The "
4901     "generalization hierarchy is a rooted, directed graph and "
4902     "does not support multiple inheritance.\n"
4903     "A class may have methods, which represent their behavior, "
4904     "and properties, which represent the data structure of its "
4905     "instances.\n"
4906     "A class may participate in associations as the target of a "
4907     "reference owned by the association.\n"
4908     "A class may have instances.") ]
4909 class Meta_Class : Meta_NamedElement
4910 {
4911     [Override ("Name"), Description (
4912         "The name of the class.\n"
4913         "The class name shall follow the formal syntax defined by "
4914         "the className ABNF rule in ANNEX A. The name of "
4915         "the schema containing the class is part of the class "
4916         "name.\n"
4917         "Class names shall be compared case insensitively.\n"
4918         "The class name shall be unique within the schema owning "
4919         "the class.") ]
4920     string Name;
4921 };
4922
4923 // =====
4924 //     Property
4925 // =====
4926 [Version("2.6.0"), Description (
4927     "Models a CIM property defined in a CIM class. A CIM property "
4928     "is the declaration of a structural feature of a CIM class, "
4929     "i.e. the data structure of its instances.\n"
4930     "Properties are inherited to subclasses such that instances of "
4931     "the subclasses have the inherited properties in addition to "
4932     "the properties defined in the subclass. The combined set of "
4933     "properties defined in a class and properties inherited from "
4934     "superclasses is called the properties exposed by the class.\n"
4935     "A class defining a property may indicate that the property "
4936     "overrides an inherited property. In this case, the class "
4937     "exposes only the overriding property. The characteristics of "
4938     "the overriding property are formed by using the "
4939     "characteristics of the overridden property as a basis, "
4940     "changing them as defined in the overriding property, within "
4941     "certain limits as defined in additional constraints.\n"
4942     "The class owning an overridden property shall be a (direct "
4943     "or indirect) superclass of the class owning the overriding "

```

```

4944 "property.\n"
4945 "For references, the class referenced by the overriding "
4946 "reference shall be the same as, or a subclass of, the class "
4947 "referenced by the overridden reference."),
4948 ClassConstraint {
4949 "/* An overriding property shall have the same name as the "
4950 "property it overrides. */\n"
4951 "inv: self.Meta_PropertyOverride[OverridingProperty]->\n"
4952 "    size() = 1\n"
4953 "    implies\n"
4954 "    self.Meta_PropertyOverride[OverridingProperty].\n"
4955 "        OverriddenProperty.Name.toUpper() =\n"
4956 "        self.Name.toUpper()",
4957 "/* For ordinary properties, the data type of the overriding "
4958 "property shall be the same as the data type of the overridden "
4959 "property. */\n"
4960 "inv: self.oclIsTypeOf(Meta_Property) and\n"
4961 "    Meta_PropertyOverride[OverridingProperty]->\n"
4962 "    size() = 1\n"
4963 "    implies\n"
4964 "    let pt : Meta_Type = /* type of property */\n"
4965 "        self.Meta_ElementType[Element].Type\n"
4966 "    in\n"
4967 "    let opt : Meta_Type = /* type of overridden prop. */\n"
4968 "        self.Meta_PropertyOverride[OverridingProperty].\n"
4969 "        OverriddenProperty.Meta_ElementType[Element].Type\n"
4970 "    in\n"
4971 "    opt.TypeName.toUpper() = pt.TypeName.toUpper() and\n"
4972 "    opt.IsArray = pt.IsArray and\n"
4973 "    opt.ArraySize = pt.ArraySize"} ]
4974 class Meta_Property : Meta_TypedElement
4975 {
4976     [Override ("Name"), Description (
4977     "The name of the property. The property name shall follow "
4978     "the formal syntax defined by the propertyName ABNF rule "
4979     "in ANNEX A.\n"
4980     "Property names shall be compared case insensitively.\n"
4981     "Property names shall be unique within its owning (i.e. "
4982     "defining) class.\n"
4983     "NOTE: The set of properties exposed by a class may have "
4984     "duplicate names if a class defines a property with the "
4985     "same name as a property it inherits without overriding "
4986     "it.") ]
4987     string Name;
4988
4989     [Description (
4990     "The default value of the property, in its string "
4991     "representation.") ]
4992     string DefaultValue [];

```

```

4993 };
4994
4995 // =====
4996 //     Method
4997 // =====
4998
4999     [Version("2.6.0"), Description (
5000         "Models a CIM method. A CIM method is the declaration of a "
5001         "behavioral feature of a CIM class, representing the ability "
5002         "for invoking an associated behavior.\n"
5003         "The CIM data type of the method defines the declared return "
5004         "type of the method.\n"
5005         "Methods are inherited to subclasses such that subclasses have "
5006         "the inherited methods in addition to the methods defined in "
5007         "the subclass. The combined set of methods defined in a class "
5008         "and methods inherited from superclasses is called the methods "
5009         "exposed by the class.\n"
5010         "A class defining a method may indicate that the method "
5011         "overrides an inherited method. In this case, the class exposes "
5012         "only the overriding method. The characteristics of the "
5013         "overriding method are formed by using the characteristics of "
5014         "the overridden method as a basis, changing them as defined in "
5015         "the overriding method, within certain limits as defined in "
5016         "additional constraints.\n"
5017         "The class owning an overridden method shall be a superclass "
5018         "of the class owning the overriding method."),
5019     ClassConstraint {
5020         /* An overriding method shall have the same name as the "
5021         "method it overrides. */\n"
5022         "inv: self.Meta_MethodOverride[OverridingMethod]->\n"
5023         "     size() = 1\n"
5024         "     implies\n"
5025         "     self.Meta_MethodOverride[OverridingMethod].\n"
5026         "     OverriddenMethod.Name.toUpper() =\n"
5027         "     self.Name.toUpper()",
5028         /* The return type of a method shall not be an array. */\n"
5029         "inv: self.Meta_ElementType[Element].Type.IsArray = false",
5030         /* An overriding method shall have the same signature "
5031         "(i.e. parameters and return type) as the method it "
5032         "overrides. */\n"
5033         "inv: Meta_MethodOverride[OverridingMethod]->size() = 1\n"
5034         "     implies\n"
5035         "     let om : Meta_Method = /* overridden method */\n"
5036         "     self.Meta_MethodOverride[OverridingMethod].\n"
5037         "     OverriddenMethod\n"
5038         "     in\n"
5039         "     om.Meta_ElementType[Element].Type.TypeName.toUpper() =\n"
5040         "     self.Meta_ElementType[Element].Type.TypeName.toUpper()\n"
5041         "     and\n"

```

```

5042     "      Set {1 .. om.Meta_MethodParameter[OwningMethod].\n"
5043         "      OwnedParameter->size()}\n"
5044     "      ->forall( i | \n"
5045         "      let omp : Meta_Parameter = /* parm in overridden method */\n"
5046         "      om.Meta_MethodParameter[OwningMethod].OwnedParameter->\n"
5047         "      asOrderedSet()->at(i)\n"
5048         "      in\n"
5049         "      let selfp : Meta_Parameter = /* parm in overriding method */\n"
5050         "      self.Meta_MethodParameter[OwningMethod].OwnedParameter->\n"
5051         "      asOrderedSet()->at(i)\n"
5052         "      in\n"
5053         "      omp.Name.toUpper() = selfp.Name.toUpper() and\n"
5054         "      omp.Meta_ElementType[Element].Type.TypeName.toUpper() =\n"
5055         "      selfp.Meta_ElementType[Element].Type.TypeName.toUpper()\n"
5056         "      )"} ]
5057 class Meta_Method : Meta_TypedElement
5058 {
5059     [Override ("Name"), Description (
5060         "The name of the method. The method name shall follow "
5061         "the formal syntax defined by the methodName ABNF rule in "
5062         "ANNEX A.\n"
5063         "Method names shall be compared case insensitively.\n"
5064         "Method names shall be unique within its owning (i.e. "
5065         "defining) class.\n"
5066         "NOTE: The set of methods exposed by a class may have "
5067         "duplicate names if a class defines a method with the same "
5068         "name as a method it inherits without overriding it.") ]
5069     string Name;
5070 };
5071
5072 // =====
5073 //     Parameter
5074 // =====
5075 [Version("2.6.0"), Description (
5076     "Models a CIM parameter. A CIM parameter is the declaration of "
5077     "a parameter of a CIM method. The return value of a "
5078     "method is not modeled as a parameter.") ]
5079 class Meta_Parameter : Meta_TypedElement
5080 {
5081     [Override ("Name"), Description (
5082         "The name of the parameter. The parameter name shall follow "
5083         "the formal syntax defined by the parameterName ABNF rule "
5084         "in ANNEX A.\n"
5085         "Parameter names shall be compared case insensitively.") ]
5086     string Name;
5087 };
5088
5089 // =====
5090 //     Trigger

```

```

5091 // =====
5092
5093 [Version("2.6.0"), Description (
5094 "Models a CIM trigger. A CIM trigger is the specification of a "
5095 "rule on a CIM element that defines when the trigger is to be "
5096 "fired.\n"
5097 "Triggers may be fired on the following occasions:\n"
5098 "* On creation, deletion, modification, or access of CIM "
5099 "instances of ordinary classes and associations. The trigger is "
5100 "specified on the class in this case and applies to all "
5101 "instances.\n"
5102 "* On modification, or access of a CIM property. The trigger is "
5103 "specified on the property in this case and and applies to all "
5104 "instances.\n"
5105 "* Before and after the invocation of a CIM method. The trigger "
5106 "is specified on the method in this case and and applies to all "
5107 "invocations of the method.\n"
5108 "* When a CIM indication is raised. The trigger is specified on "
5109 "the indication in this case and and applies to all occurrences "
5110 "for when this indication is raised.\n"
5111 "The rules for when a trigger is to be fired are specified with "
5112 "the TriggerType qualifier.\n"
5113 "The firing of a trigger shall cause the indications to be "
5114 "raised that are associated to the trigger via "
5115 "Meta_TriggeredIndication."),
5116 ClassConstraint {
5117 /* Triggers shall be specified only on ordinary classes, "
5118 "associations, properties (including references), methods and "
5119 "indications. */\n"
5120 "inv: let e : Meta_NamedElement = /* the element on which\n"
5121 "           the trigger is specified */\n"
5122 "       self.Meta_TriggeringElement[Trigger].Element\n"
5123 "       in\n"
5124 "       e.oclIsTypeOf(Meta_Class) or\n"
5125 "       e.oclIsTypeOf(Meta_Association) or\n"
5126 "       e.oclIsTypeOf(Meta_Property) or\n"
5127 "       e.oclIsTypeOf(Meta_Reference) or\n"
5128 "       e.oclIsTypeOf(Meta_Method) or\n"
5129 "       e.oclIsTypeOf(Meta_Indication)} ]
5130 class Meta_Trigger : Meta_NamedElement
5131 {
5132     [Override ("Name"), Description (
5133     "The name of the trigger.\n"
5134     "Trigger names shall be compared case insensitively.\n"
5135     "Trigger names shall be unique "
5136     "within the property, class or method to which the trigger "
5137     "applies.") ]
5138     string Name;
5139 };

```

```

5140
5141 // =====
5142 //      Indication
5143 // =====
5144
5145     [Version("2.6.0"), Description (
5146     "Models a CIM indication. An instance of a CIM indication "
5147     "represents an event that has occurred. If an instance of an "
5148     "indication is created, the indication is said to be raised. "
5149     "The event causing an indication to be raised may be that a "
5150     "trigger has fired, but other arbitrary events may cause an "
5151     "indication to be raised as well."),
5152     ClassConstraint {
5153     "/* An indication shall not own any methods. */\n"
5154     "inv: self.MethodDomain[OwningClass].OwnedMethod->size() = 0" } ]
5155 class Meta_Indication : Meta_Class
5156 {
5157 };
5158
5159 // =====
5160 //      Association
5161 // =====
5162
5163     [Version("2.6.0"), Description (
5164     "Models a CIM association. A CIM association is a special kind "
5165     "of CIM class that represents a relationship between two or more "
5166     "CIM classes. A CIM association owns its association ends (i.e. "
5167     "references). This allows for adding associations to a schema "
5168     "without affecting the associated classes."),
5169     ClassConstraint {
5170     "/* The superclass of an association shall be an association. */\n"
5171     "inv: self.Meta_Generalization[SubClass].SuperClass->\n"
5172     "    oclIsTypeOf(Meta_Association)",
5173     "/* An association shall own two or more references. */\n"
5174     "inv: self.Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
5175     "    select( p | p.oclIsTypeOf(Meta_Reference))->size() >= 2",
5176     "/* The number of references exposed by an association (i.e. "
5177     "its arity) shall not change in its subclasses. */\n"
5178     "inv: self.Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
5179     "    select( p | p.oclIsTypeOf(Meta_Reference))->size() =\n"
5180     "    self.Meta_Generalization[SubClass].SuperClass->\n"
5181     "    Meta_PropertyDomain[OwningClass].OwnedProperty->\n"
5182     "    select( p | p.oclIsTypeOf(Meta_Reference))->size()"} ]
5183 class Meta_Association : Meta_Class
5184 {
5185 };
5186
5187 // =====
5188 //      Reference

```

```

5189 // =====
5190
5191 [Version("2.6.0"), Description (
5192 "Models a CIM reference. A CIM reference is a special kind of "
5193 "CIM property that represents an association end, as well as a "
5194 "role the referenced class plays in the context of the "
5195 "association owning the reference."),
5196 ClassConstraint {
5197 "/* A reference shall be owned by an association (i.e. not "
5198 "by an ordinary class or by an indication). As a result "
5199 "of this, reference names do not need to be unique within any "
5200 "of the associated classes. */\n"
5201 "inv: self.Meta_PropertyDomain[OwnedProperty].OwningClass.\n"
5202 "    oclIsTypeOf(Meta_Association)"} ]
5203 class Meta_Reference : Meta_Property
5204 {
5205     [Override ("Name"), Description (
5206 "The name of the reference. The reference name shall follow "
5207 "the formal syntax defined by the referenceName ABNF rule "
5208 "in ANNEX A.\n"
5209 "Reference names shall be compared case insensitively.\n"
5210 "Reference names shall be unique within its owning (i.e. "
5211 "defining) association.") ]
5212     string Name;
5213 };
5214
5215 // =====
5216 //     QualifierType
5217 // =====
5218 [Version("2.6.0"), Description (
5219 "Models the declaration of a CIM qualifier (i.e. a qualifier "
5220 "type). A CIM qualifier is meta data that provides additional "
5221 "information about the element on which the qualifier is "
5222 "specified.\n"
5223 "The qualifier type is either explicitly defined in the CIM "
5224 "namespace, or implicitly defined on an element as a result of "
5225 "a qualifier that is specified on an element for which no "
5226 "explicit qualifier type is defined.\n"
5227 "Implicitly defined qualifier types shall agree in data type, "
5228 "scope, flavor and default value with any explicitly defined "
5229 "qualifier types of the same name. \n"
5230 "DEPRECATED: The concept of implicitly defined qualifier "
5231 "types is deprecated.") ]
5232 class Meta_QualifierType : Meta_TypedElement
5233 {
5234     [Override ("Name"), Description (
5235 "The name of the qualifier. The qualifier name shall follow "
5236 "the formal syntax defined by the qualifierName ABNF rule "
5237 "in ANNEX A.\n"

```

```

5238     "The names of explicitly defined qualifier types shall be "
5239     "unique within the CIM namespace. Unlike classes, "
5240     "qualifier types are not part of a schema, so name "
5241     "uniqueness cannot be defined at the definition level "
5242     "relative to a schema, and is instead only defined at "
5243     "the object level relative to a namespace.\n"
5244     "The names of implicitly defined qualifier types shall be "
5245     "unique within the scope of the CIM element on which the "
5246     "qualifiers are specified.") ]
5247 string Name;
5248
5249     [Description (
5250     "The scopes of the qualifier. The qualifier scopes determine "
5251     "to which kinds of elements a qualifier may be specified on. "
5252     "Each qualifier scope shall be one of the following keywords:\n"
5253     "  \"any\" - the qualifier may be specified on any qualifiable element.\n"
5254     "  \"class\" - the qualifier may be specified on any ordinary class.\n"
5255     "  \"association\" - the qualifier may be specified on any association.\n"
5256     "  \"indication\" - the qualifier may be specified on any indication.\n"
5257     "  \"property\" - the qualifier may be specified on any ordinary property.\n"
5258     "  \"reference\" - the qualifier may be specified on any reference.\n"
5259     "  \"method\" - the qualifier may be specified on any method.\n"
5260     "  \"parameter\" - the qualifier may be specified on any parameter.\n"
5261     "Qualifiers cannot be specified on qualifiers.") ]
5262 string Scope [];
5263 };
5264
5265 // =====
5266 //   Qualifier
5267 // =====
5268
5269     [Version("2.6.0"), Description (
5270     "Models the specification (i.e. usage) of a CIM qualifier on an "
5271     "element. A CIM qualifier is meta data that provides additional "
5272     "information about the element on which the qualifier is "
5273     "specified. The specification of a qualifier on an element "
5274     "defines a value for the qualifier on that element.\n"
5275     "If no explicitly defined qualifier type exists with this name "
5276     "in the CIM namespace, the specification of a qualifier causes an "
5277     "implicitly defined qualifier type (i.e. a Meta_QualifierType "
5278     "element) to be created on the qualified element. \n"
5279     "DEPRECATED: The concept of implicitly defined qualifier "
5280     "types is deprecated.") ]
5281 class Meta_Qualifier : Meta_NamedElement
5282 {
5283     [Override ("Name"), Description (
5284     "The name of the qualifier. The qualifier name shall follow "
5285     "the formal syntax defined by the qualifierName ABNF rule "
5286     "in ANNEX A. \n

```

```

5287     "The names of explicitly defined qualifier types shall be "
5288     "unique within the CIM namespace. Unlike classes, "
5289     "qualifier types are not part of a schema, so name "
5290     "uniqueness cannot be defined at the definition level "
5291     "relative to a schema, and is instead only defined at "
5292     "the object level relative to a namespace.\n"
5293     "The names of implicitly defined qualifier types shall be "
5294     "unique within the scope of the CIM element on which the "
5295     "qualifiers are specified." \n
5296     "DEPRECATED: The concept of implicitly defined qualifier "
5297     "types is deprecated.") ]
5298     string Name;
5299
5300     [Description (
5301     "The scopes of the qualifier. The qualifier scopes determine "
5302     "to which kinds of elements a qualifier may be specified on. "
5303     "Each qualifier scope shall be one of the following keywords:\n"
5304     "  \"any\" - the qualifier may be specified on any qualifiable element.\n"
5305     "  \"class\" - the qualifier may be specified on any ordinary class.\n"
5306     "  \"association\" - the qualifier may be specified on any association.\n"
5307     "  \"indication\" - the qualifier may be specified on any indication.\n"
5308     "  \"property\" - the qualifier may be specified on any ordinary property.\n"
5309     "  \"reference\" - the qualifier may be specified on any reference.\n"
5310     "  \"method\" - the qualifier may be specified on any method.\n"
5311     "  \"parameter\" - the qualifier may be specified on any parameter.\n"
5312     "Qualifiers cannot be specified on qualifiers.") ]
5313     string Scope [];
5314 };
5315
5316 // =====
5317 //     Flavor
5318 // =====
5319     [Version("2.6.0"), Description (
5320     "The specification of certain characteristics of the qualifier "
5321     "such as its value propagation from the ancestry of the "
5322     "qualified element, and translatability of the qualifier "
5323     "value.") ]
5324 class Meta_Flavor
5325 {
5326     [Description (
5327     "Indicates whether the qualifier value is to be propagated "
5328     "from the ancestry of an element in case the qualifier is "
5329     "not specified on the element.") ]
5330     boolean InheritancePropagation;
5331
5332     [Description (
5333     "Indicates whether qualifier values propagated to an "
5334     "element may be overridden by the specification of that "
5335     "qualifier on the element.") ]

```

```

5336     boolean OverridePermission;
5337
5338         [Description (
5339             "Indicates whether qualifier value is translatable.") ]
5340     boolean Translatable;
5341 };
5342
5343 // =====
5344 //     Instance
5345 // =====
5346     [Version("2.6.0"), Description (
5347         "Models a CIM instance. A CIM instance is an instance of a CIM "
5348         "class that specifies values for a subset (including all) of the "
5349         "properties exposed by its defining class.\n"
5350         "A CIM instance in a CIM server shall have exactly the properties "
5351         "exposed by its defining class.\n"
5352         "A CIM instance cannot redefine the properties "
5353         "or methods exposed by its defining class and cannot have "
5354         "qualifiers specified.\n"
5355         "A particular property shall be specified at most once in a "
5356         "given instance.") ]
5357 class Meta_Instance
5358 {
5359 };
5360
5361 // =====
5362 //     InstanceProperty
5363 // =====
5364     [Version("2.6.0"), Description (
5365         "The definition of a property value within a CIM instance.") ]
5366 class Meta_InstanceProperty
5367 {
5368 };
5369
5370 // =====
5371 //     Value
5372 // =====
5373     [Version("2.6.0"), Description (
5374         "A typed value, used in several contexts."),
5375     ClassConstraint {
5376         "/* If the NULL indicator is set, no values shall be specified. "
5377         "*/\n"
5378         "inv: self.IsNull = true\n"
5379         "    implies self.Value->size() = 0",
5380         "/* If values are specified, the NULL indicator shall not be "
5381         "set. */\n"
5382         "inv: self.Value->size() > 0\n"
5383         "    implies self.IsNull = false",
5384         "/* A Value instance shall be owned by only one owner. */\n"

```

```

5385     "inv: self.OwningProperty->size() +\n"
5386     "     self.OwningInstanceProperty->size() +\n"
5387     "     self.OwningQualifierType->size() +\n"
5388     "     self.OwningQualifier->size() = 1"} ]
5389 class Meta_Value
5390 {
5391     [Description (
5392         "The scalar value or the array of values. "
5393         "Each value is represented as a string.") ]
5394     string Value [];
5395
5396     [Description (
5397         "The NULL indicator of the value. "
5398         "If true, the value is NULL. "
5399         "If false, the value is indicated through the Value "
5400         "attribute.") ]
5401     boolean IsNull;
5402 };
5403
5404 // =====
5405 //     SpecifiedQualifier
5406 // =====
5407     [Association, Composition, Version("2.6.0")]
5408 class Meta_SpecifiedQualifier
5409 {
5410     [Aggregate, Min (1), Max (1), Description (
5411         "The element on which the qualifier is specified.") ]
5412     Meta_NamedElement REF OwningElement;
5413
5414     [Min (0), Max (NULL), Description (
5415         "The qualifier specified on the element.") ]
5416     Meta_Qualifier REF OwnedQualifier;
5417 };
5418
5419 // =====
5420 //     ElementType
5421 // =====
5422     [Association, Composition, Version("2.6.0")]
5423 class Meta_ElementType
5424 {
5425     [Aggregate, Min (0), Max (1), Description (
5426         "The element that has a CIM data type.") ]
5427     Meta_TypedElement REF OwningElement;
5428
5429     [Min (1), Max (1), Description (
5430         "The CIM data type of the element.") ]
5431     Meta_Type REF OwnedType;
5432 };
5433

```

```
5434 // =====
5435 //   PropertyDomain
5436 // =====
5437
5438     [Association, Composition, Version("2.6.0")]
5439 class Meta_PropertyDomain
5440 {
5441     [Aggregate, Min (1), Max (1), Description (
5442     "The class owning (i.e. defining) the property.") ]
5443     Meta_Class REF OwningClass;
5444
5445     [Min (0), Max (NULL), Description (
5446     "The property owned by the class.") ]
5447     Meta_Property REF OwnedProperty;
5448 };
5449
5450 // =====
5451 //   MethodDomain
5452 // =====
5453
5454     [Association, Composition, Version("2.6.0")]
5455 class Meta_MethodDomain
5456 {
5457     [Aggregate, Min (1), Max (1), Description (
5458     "The class owning (i.e. defining) the method.") ]
5459     Meta_Class REF OwningClass;
5460
5461     [Min (0), Max (NULL), Description (
5462     "The method owned by the class.") ]
5463     Meta_Method REF OwnedMethod;
5464 };
5465
5466 // =====
5467 //   ReferenceRange
5468 // =====
5469
5470     [Association, Version("2.6.0")]
5471 class Meta_ReferenceRange
5472 {
5473     [Min (0), Max (NULL), Description (
5474     "The reference type referencing the class.") ]
5475     Meta_ReferenceType REF ReferencingType;
5476
5477     [Min (1), Max (1), Description (
5478     "The class referenced by the reference type.") ]
5479     Meta_Class REF ReferencedClass;
5480 };
5481
5482 // =====
```

```

5483 //      QualifierTypeFlavor
5484 // =====
5485
5486     [Association, Composition, Version("2.6.0")]
5487 class Meta_QualifierTypeFlavor
5488 {
5489     [Aggregate, Min (1), Max (1), Description (
5490     "The qualifier type defining the flavor.") ]
5491     Meta_QualifierType REF QualifierType;
5492
5493     [Min (1), Max (1), Description (
5494     "The flavor of the qualifier type.") ]
5495     Meta_Flavor REF Flavor;
5496 };
5497
5498 // =====
5499 //      Generalization
5500 // =====
5501
5502     [Association, Version("2.6.0")]
5503 class Meta_Generalization
5504 {
5505     [Min (0), Max (NULL), Description (
5506     "The subclass of the class.") ]
5507     Meta_Class REF SubClass;
5508
5509     [Min (0), Max (1), Description (
5510     "The superclass of the class.") ]
5511     Meta_Class REF SuperClass;
5512 };
5513
5514 // =====
5515 //      PropertyOverride
5516 // =====
5517
5518     [Association, Version("2.6.0")]
5519 class Meta_PropertyOverride
5520 {
5521     [Min (0), Max (NULL), Description (
5522     "The property overriding this property.") ]
5523     Meta_Property REF OverridingProperty;
5524
5525     [Min (0), Max (1), Description (
5526     "The property overridden by this property.") ]
5527     Meta_Property REF OverriddenProperty;
5528 };
5529
5530 // =====
5531 //      MethodOverride

```

```

5532 // =====
5533
5534     [Association, Version("2.6.0")]
5535 class Meta_MethodOverride
5536 {
5537     [Min (0), Max (NULL), Description (
5538     "The method overriding this method.") ]
5539     Meta_Method REF OverridingMethod;
5540
5541     [Min (0), Max (1), Description (
5542     "The method overridden by this method.") ]
5543     Meta_Method REF OverriddenMethod;
5544 };
5545
5546 // =====
5547 //     SchemaElement
5548 // =====
5549
5550     [Association, Composition, Version("2.6.0")]
5551 class Meta_SchemaElement
5552 {
5553     [Aggregate, Min (1), Max (1), Description (
5554     "The schema owning the element.") ]
5555     Meta_Schema REF OwningSchema;
5556
5557     [Min (0), Max (NULL), Description (
5558     "The elements owned by the schema.") ]
5559     Meta_NamedElement REF OwnedElement;
5560 };
5561
5562 // =====
5563 //     MethodParameter
5564 // =====
5565     [Association, Composition, Version("2.6.0")]
5566 class Meta_MethodParameter
5567 {
5568     [Aggregate, Min (1), Max (1), Description (
5569     "The method owning (i.e. defining) the parameter.") ]
5570     Meta_Method REF OwningMethod;
5571
5572     [Min (0), Max (NULL), Description (
5573     "The parameter of the method. The return value "
5574     "is not represented as a parameter.") ]
5575     Meta_Parameter REF OwnedParameter;
5576 };
5577
5578 // =====
5579 //     SpecifiedProperty
5580 // =====

```

```

5581     [Association, Composition, Version("2.6.0")]
5582 class Meta_SpecifiedProperty
5583 {
5584     [Aggregate, Min (1), Max (1), Description (
5585     "The instance for which a property value is defined.") ]
5586     Meta_Instance REF OwningInstance;
5587
5588     [Min (0), Max (NULL), Description (
5589     "The property value specified by the instance.") ]
5590     Meta_PropertyValue REF OwnedPropertyValue;
5591 };
5592
5593 // =====
5594 //     DefiningClass
5595 // =====
5596     [Association, Version("2.6.0")]
5597 class Meta_DefiningClass
5598 {
5599     [Min (0), Max (NULL), Description (
5600     "The instances for which the class is their defining class.") ]
5601     Meta_Instance REF Instance;
5602
5603     [Min (1), Max (1), Description (
5604     "The defining class of the instance.") ]
5605     Meta_Class REF DefiningClass;
5606 };
5607
5608 // =====
5609 //     DefiningQualifier
5610 // =====
5611     [Association, Version("2.6.0")]
5612 class Meta_DefiningQualifier
5613 {
5614     [Min (0), Max (NULL), Description (
5615     "The specification (i.e. usage) of the qualifier.") ]
5616     Meta_Qualifier REF Qualifier;
5617
5618     [Min (1), Max (1), Description (
5619     "The qualifier type defining the characteristics of the "
5620     "qualifier.") ]
5621     Meta_QualifierType REF QualifierType;
5622 };
5623
5624 // =====
5625 //     DefiningProperty
5626 // =====
5627     [Association, Version("2.6.0")]
5628 class Meta_DefiningProperty
5629 {

```

```

5630     [Min (1), Max (1), Description (
5631         "A value of this property in an instance.") ]
5632     Meta_PropertyValue REF InstanceProperty;
5633
5634     [Min (0), Max (NULL), Description (
5635         "The declaration of the property for which a value is "
5636         "defined.") ]
5637     Meta_Property REF DefiningProperty;
5638 };
5639
5640 // =====
5641 //     ElementQualifierType
5642 // =====
5643     [Association, Version("2.6.0"), Description (
5644         "DEPRECATED: The concept of implicitly defined qualifier "
5645         "types is deprecated.") ]
5646 class Meta_ElementQualifierType
5647 {
5648     [Min (0), Max (1), Description (
5649         "For implicitly defined qualifier types, the element on "
5650         "which the qualifier type is defined.\n"
5651         "Qualifier types defined explicitly are not "
5652         "associated to elements, they are global in the CIM "
5653         "namespace.") ]
5654     Meta_NamedElement REF Element;
5655
5656     [Min (0), Max (NULL), Description (
5657         "The qualifier types implicitly defined on the element.\n"
5658         "Qualifier types defined explicitly are not "
5659         "associated to elements, they are global in the CIM "
5660         "namespace.") ]
5661     Meta_QualifierType REF QualifierType;
5662 };
5663
5664 // =====
5665 //     TriggeringElement
5666 // =====
5667     [Association, Version("2.6.0")]
5668 class Meta_TriggeringElement
5669 {
5670     [Min (0), Max (NULL), Description (
5671         "The triggers specified on the element.") ]
5672     Meta_Trigger REF Trigger;
5673
5674     [Min (1), Max (NULL), Description (
5675         "The CIM element on which the trigger is specified.") ]
5676     Meta_NamedElement REF Element;
5677 };
5678

```

```

5679 // =====
5680 //   TriggeredIndication
5681 // =====
5682   [Association, Version("2.6.0")]
5683 class Meta_TriggeredIndication
5684 {
5685     [Min (0), Max (NULL), Description (
5686         "The triggers specified on the element.") ]
5687     Meta_Trigger REF Trigger;
5688
5689     [Min (0), Max (NULL), Description (
5690         "The CIM element on which the trigger is specified.") ]
5691     Meta_Indication REF Indication;
5692 };
5693 // =====
5694 //   ValueType
5695 // =====
5696   [Association, Composition, Version("2.6.0")]
5697 class Meta_ValueType
5698 {
5699     [Aggregate, Min (0), Max (1), Description (
5700         "The value that has a CIM data type.") ]
5701     Meta_Value REF OwningValue;
5702
5703     [Min (1), Max (1), Description (
5704         "The type of this value.") ]
5705     Meta_Type REF OwnedType;
5706 };
5707
5708 // =====
5709 //   PropertyDefaultValue
5710 // =====
5711   [Association, Composition, Version("2.6.0")]
5712 class Meta_PropertyDefaultValue
5713 {
5714     [Aggregate, Min (0), Max (1), Description (
5715         "A property declaration that defines this value as its "
5716         "default value.") ]
5717     Meta_Property REF OwningProperty;
5718
5719     [Min (0), Max (1), Description (
5720         "The default value of the property declaration. A Value "
5721         "instance shall be associated if and only if a default "
5722         "value is defined on the property declaration.") ]
5723     Meta_Value REF OwnedDefaultValue;
5724 };
5725
5726 // =====
5727 //   QualifierTypeDefaultValue

```

```

5728 // =====
5729     [Association, Composition, Version("2.6.0")]
5730 class Meta_QualifierTypeDefaultValue
5731 {
5732     [Aggregate, Min (0), Max (1), Description (
5733     "A qualifier type declaration that defines this value as "
5734     "its default value.") ]
5735     Meta_QualifierType REF OwningQualifierType;
5736
5737     [Min (0), Max (1), Description (
5738     "The default value of the qualifier declaration. A Value "
5739     "instance shall be associated if and only if a default "
5740     "value is defined on the qualifier declaration.") ]
5741     Meta_Value REF OwnedDefaultValue;
5742 };
5743
5744 // =====
5745 //     PropertyValue
5746 // =====
5747     [Association, Composition, Version("2.6.0")]
5748 class Meta_PropertyValue
5749 {
5750     [Aggregate, Min (0), Max (1), Description (
5751     "A property defined in an instance that has this value.") ]
5752     Meta_InstanceProperty REF OwningInstanceProperty;
5753
5754     [Min (1), Max (1), Description (
5755     "The value of the property.") ]
5756     Meta_Value REF OwnedValue;
5757
5758 // =====
5759 //     QualifierValue
5760 // =====
5761     [Association, Composition, Version("2.6.0")]
5762 class Meta_QualifierValue
5763 {
5764     [Aggregate, Min (0), Max (1), Description (
5765     "A qualifier defined on a schema element that has this "
5766     "value.") ]
5767     Meta_Qualifier REF OwningQualifier;
5768
5769     [Min (1), Max (1), Description (
5770     "The value of the qualifier.") ]
5771     Meta_Value REF OwnedValue;
5772 };

```

ANNEX C (normative)

Units

5773
5774
5775
5776

5777 C.1 Programmatic Units

5778 This annex defines the concept and syntax of a programmatic unit, which is an expression of a unit of
5779 measure for programmatic access. It makes it easy to recognize the base units of which the actual unit is
5780 made, as well as any numerical multipliers. Programmatic units are used as a value for the PUnit qualifier
5781 and also as a value for any (string typed) CIM elements that represent units. The boolean IsPUnit qualifier
5782 is used to declare that a string typed element follows the syntax for programmatic units.

5783 Programmatic units must be processed case-sensitively and white-space-sensitively.

5784 As defined in the Augmented BNF (ABNF) syntax, the programmatic unit consists of a base unit that is
5785 optionally followed by other base units that are each either multiplied or divided into the first base unit.
5786 Furthermore, two optional multipliers can be applied. The first is simply a scalar, and the second is an
5787 exponential number consisting of a base and an exponent. The optional multipliers enable the
5788 specification of common derived units of measure in terms of the allowed base units. The base units
5789 defined in this subclause include a superset of the SI base units. When a unit is the empty string, the
5790 value has no unit; that is, it is dimensionless. The multipliers must be understood as part of the definition
5791 of the derived unit; that is, scale prefixes of units are replaced with their numerical value. For example,
5792 "kilometer" is represented as "meter * 1000", replacing the "kilo" scale prefix with the numerical factor
5793 1000.

5794 A string representing a programmatic unit must follow the format defined by the `programmatic-unit`
5795 ABNF rule in the syntax defined in this annex. This format supports any type of unit, including SI units,
5796 United States units, and any other standard or non-standard units.

5797 The ABNF syntax is defined as follows. This ABNF explicitly states any whitespace characters that may
5798 be used, and whitespace characters in addition to those are not allowed.

```
5799 programmatic-unit = ( "" / base-unit *( [WS] multiplied-base-unit )
5800                      *( [WS] divided-base-unit ) [ [WS] modifier1 ] [ [WS] modifier2 ] )
5801
5802 multiplied-base-unit = "*" [WS] base-unit
5803
5804 divided-base-unit = "/" [WS] base-unit
5805
5806 modifier1 = operator [WS] number
5807
5808 modifier2 = operator [WS] base [WS] "^" [WS] exponent
5809
5810 operator = "*" / "/"
5811
5812 number = ["+" / "-"] positive-number
5813
5814 base = positive-whole-number
5815
5816 exponent = ["+" / "-"] positive-whole-number
```

```

5817
5818 positive-whole-number = NON-ZERO-DIGIT *( DIGIT )
5819
5820 positive-number = positive-whole-number
5821                   / ( ( positive-whole-number / ZERO ) "." *( DIGIT ) )
5822
5823 base-unit = simple-name / decibel-base-unit
5824
5825 simple-name = FIRST-UNIT-CHAR *( [S] UNIT-CHAR )
5826
5827 decibel-base-unit = "decibel" [ [S] "(" [S] simple-name [S] ")" ]
5828
5829 FIRST-UNIT-CHAR = UPPERALPHA / LOWERALPHA / UNDERSCORE / UCS0080TOFFEF
5830                   ; DEPRECATED: The use of the UCS0080TOFFEF ABNF rule within
5831                   ; the FIRST-UNIT-CHAR ABNF rule is deprecated since
5832                   ; version 2.6.0 of this document.
5833
5834 UNIT-CHAR = FIRST-UNIT-CHAR / S / HYPHEN / DIGIT
5835
5836 ZERO = "0"
5837
5838 NON-ZERO-DIGIT = ("1"..."9")
5839
5840 DIGIT = ZERO / NON-ZERO-DIGIT
5841
5842 WS = ( S / TAB / NL )
5843
5844 S = U+0020           ; " " (space)
5845
5846 TAB = U+0009        ; "\t" (tab)
5847
5848 NL = U+000A         ; "\n" (newline, linefeed)
5849
5850 HYPHEN = U+000A     ; "-" (hyphen, minus)

```

5851 The ABNF rules UPPERALPHA, LOWERALPHA, UNDERSCORE, UCS0080TOFFEF are defined in
5852 ANNEX A.

5853 For example, a speedometer may be modeled so that the unit of measure is kilometers per hour. It is
5854 necessary to express the derived unit of measure "kilometers per hour" in terms of the allowed base units
5855 "meter" and "second". One kilometer per hour is equivalent to

```

5856     1000 meters per 3600 seconds
5857     or
5858     one meter / second / 3.6

```

5859 so the programmatic unit for "kilometers per hour" is expressed as: "meter / second / 3.6", using the
5860 syntax defined here.

5861 Other examples are as follows:

- 5862 "meter * meter * 10⁻⁶" → square millimeters
- 5863 "byte * 2¹⁰" → kBytes as used for memory ("kibobyte")
- 5864 "byte * 10³" → kBytes as used for storage ("kilobyte")
- 5865 "dataword * 4" → QuadWords
- 5866 "decibel(m) * -1" → -dBm
- 5867 "second * 250 * 10⁻⁹" → 250 nanoseconds
- 5868 "foot * foot * foot / minute" → cubic feet per minute, CFM
- 5869 "revolution / minute" → revolutions per minute, RPM
- 5870 "pound / inch / inch" → pounds per square inch, PSI
- 5871 "foot * pound" → foot-pounds

5872 In the "PU Base Unit" column, Table C-1 defines the allowed values for the `base-unit` ABNF rule in the
 5873 syntax, as well as the empty string indicating no unit. The "Symbol" column recommends a symbol to be
 5874 used in a human interface. The "Calculation" column relates units to other units. The "Quantity" column
 5875 lists the physical quantity measured by the unit.

5876 The base units in Table C-1 consist of the SI base units and the SI derived units amended by other
 5877 commonly used units. "SI" is the international abbreviation for the International System of Units (French:
 5878 "Système International d'Unites"), defined in ISO 1000:1992. Also, ISO 1000:1992 defines the notational
 5879 conventions for units, which are used in Table C-1.

5880 **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

PU Base Unit	Symbol	Calculation	Quantity
rack unit	U	1 U = 1.75 in	Length (height unit used for computer components, as defined in EIA-310)
foot	ft	1 ft = 12 in	Length
yard	yd	1 yd = 3 ft	Length
mile	mi	1 mi = 1760 yd	Length (U.S. land mile)
liter	l	1000 l = 1 m ³	Volume (The corresponding ISO SI unit is "litre.")
fluid ounce	fl.oz	33.8140227 fl.oz = 1 l	Volume for liquids (U.S. fluid ounce)
liquid gallon	gal	1 gal = 128 fl.oz	Volume for liquids (U.S. liquid gallon)
mole	mol	SI base unit	Amount of substance
kilogram	kg	SI base unit	Mass
ounce	oz	35.27396195 oz = 1 kg	Mass (U.S. ounce, avoirdupois ounce)
pound	lb	1 lb = 16 oz	Mass (U.S. pound, avoirdupois pound)
second	s	SI base unit	Time (duration)
minute	min	1 min = 60 s	Time (duration)
hour	h	1 h = 60 min	Time (duration)
day	d	1 d = 24 h	Time (duration)
week	week	1 week = 7 d	Time (duration)
hertz	Hz	1 Hz = 1 /s	Frequency
gravity	g	1 g = 9.80665 m/s ²	Acceleration
degree celsius	°C	1 °C = 1 K (diff)	Thermodynamic temperature
degree fahrenheit	°F	1 °F = 5/9 K (diff)	Thermodynamic temperature
kelvin	K	SI base unit	Thermodynamic temperature, color temperature
candela	cd	SI base unit	Luminous intensity
lumen	lm	1 lm = 1 cd·sr	Luminous flux
nit	nit	1 nit = 1 cd/m ²	Luminance
lux	lx	1 lx = 1 lm/m ²	Illuminance
newton	N	1 N = 1 kg·m/s ²	Force
pascal	Pa	1 Pa = 1 N/m ²	Pressure
bar	bar	1 bar = 100000 Pa	Pressure
decibel(A)	dB(A)	1 dB(A) = 20 lg (p/p ₀)	Loudness of sound, relative to reference sound pressure level of p ₀ = 20 μPa in gases, using frequency weight curve (A)

PU Base Unit	Symbol	Calculation	Quantity
decibel(C)	dB(C)	$1 \text{ dB(C)} = 20 \cdot \lg(p/p_0)$	Loudness of sound, relative to reference sound pressure level of $p_0 = 20 \mu\text{Pa}$ in gases, using frequency weight curve (C)
joule	J	$1 \text{ J} = 1 \text{ N}\cdot\text{m}$	Energy, work, torque, quantity of heat
watt	W	$1 \text{ W} = 1 \text{ J/s} = 1 \text{ V} \cdot \text{A}$	Power, radiant flux. In electric power technology, the real power (also known as active power or effective power or true power)
volt ampere	VA	$1 \text{ VA} = 1 \text{ V} \cdot \text{A}$	In electric power technology, the apparent power
volt ampere reactive	var	$1 \text{ var} = 1 \text{ V} \cdot \text{A}$	In electric power technology, the reactive power (also known as imaginary power)
decibel(m)	dBm	$1 \text{ dBm} = 10 \cdot \lg(P/P_0)$	Power, relative to reference power of $P_0 = 1 \text{ mW}$
british thermal unit	BTU	$1 \text{ BTU} = 1055.056 \text{ J}$	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	$1 \text{ C} = 1 \text{ A}\cdot\text{s}$	Electric charge
volt	V	$1 \text{ V} = 1 \text{ W/A}$	Electric tension, electric potential, electromotive force
farad	F	$1 \text{ F} = 1 \text{ C/V}$	Capacitance
ohm	Ohm	$1 \text{ Ohm} = 1 \text{ V/A}$	Electric resistance
siemens	S	$1 \text{ S} = 1 / \text{Ohm}$	Electric conductance
weber	Wb	$1 \text{ Wb} = 1 \text{ V}\cdot\text{s}$	Magnetic flux
tesla	T	$1 \text{ T} = 1 \text{ Wb/m}^2$	Magnetic flux density, magnetic induction
henry	H	$1 \text{ H} = 1 \text{ Wb/A}$	Inductance
becquerel	Bq	$1 \text{ Bq} = 1 / \text{s}$	Activity (of a radionuclide)
gray	Gy	$1 \text{ Gy} = 1 \text{ J/kg}$	Absorbed dose, specific energy imparted, kerma, absorbed dose index
sievert	Sv	$1 \text{ Sv} = 1 \text{ J/kg}$	Dose equivalent, dose equivalent index

5881 C.2 Value for Units Qualifier

5882 DEPRECATED

5883 The Units qualifier has been used both for programmatic access and for displaying a unit. Because it
 5884 does not satisfy the full needs of either of these uses, the Units qualifier is deprecated. The PUnit qualifier
 5885 should be used instead for programmatic access.

5886 DEPRECATED

5887 For displaying a unit, the CIM client should construct the string to be displayed from the PUnit qualifier
 5888 using the conventions of the CIM client.

5889 The UNITS qualifier specifies the unit of measure in which the qualified property, method return value, or
 5890 method parameter is expressed. For example, a Size property might have Units (Bytes). The complete
 5891 set of DMTF-defined values for the Units qualifier is as follows:

- 5892 • Bits, KiloBits, MegaBits, GigaBits
- 5893 • < Bits, KiloBits, MegaBits, GigaBits> per Second
- 5894 • Bytes, KiloBytes, MegaBytes, GigaBytes, Words, DoubleWords, QuadWords
- 5895 • Degrees C, Tenths of Degrees C, Hundredths of Degrees C, Degrees F, Tenths of Degrees F,
5896 Hundredths of Degrees F, Degrees K, Tenths of Degrees K, Hundredths of Degrees K, Color
5897 Temperature
- 5898 • Volts, MilliVolts, Tenths of MilliVolts, Amps, MilliAmps, Tenths of MilliAmps, Watts,
5899 MilliWattHours
- 5900 • Joules, Coulombs, Newtons
- 5901 • Lumen, Lux, Candelas
- 5902 • Pounds, Pounds per Square Inch
- 5903 • Cycles, Revolutions, Revolutions per Minute, Revolutions per Second
- 5904 • Minutes, Seconds, Tenths of Seconds, Hundredths of Seconds, MicroSeconds, MilliSeconds,
5905 NanoSeconds
- 5906 • Hours, Days, Weeks
- 5907 • Hertz, MegaHertz
- 5908 • Pixels, Pixels per Inch
- 5909 • Counts per Inch
- 5910 • Percent, Tenths of Percent, Hundredths of Percent, Thousandths
- 5911 • Meters, Centimeters, Millimeters, Cubic Meters, Cubic Centimeters, Cubic Millimeters
- 5912 • Inches, Feet, Cubic Inches, Cubic Feet, Ounces, Liters, Fluid Ounces
- 5913 • Radians, Steradians, Degrees
- 5914 • Gravities, Pounds, Foot-Pounds
- 5915 • Gauss, Gilberts, Henrys, MilliHenrys, Farads, MilliFarads, MicroFarads, PicoFarads
- 5916 • Ohms, Siemens
- 5917 • Moles, Becquerels, Parts per Million
- 5918 • Decibels, Tenths of Decibels
- 5919 • Grays, Sieverts
- 5920 • MilliWatts
- 5921 • DBm
- 5922 • <Bytes, KiloBytes, MegaBytes, GigaBytes> per Second
- 5923 • BTU per Hour
- 5924 • PCI clock cycles
- 5925 • <Numeric value> <Minutes, Seconds, Tenths of Seconds, Hundreths of Seconds,
5926 MicroSeconds, MilliSeconds, Nanoseconds>
- 5927 • Us
- 5928 • Amps at <Numeric Value> Volts

- 5929 • Clock Ticks
- 5930 • Packets, per Thousand Packets

ANNEX D (informative)

UML Notation

5931
5932
5933
5934

5935 The CIM meta-schema notation is directly based on the notation used in Unified Modeling Language
5936 (UML). There are distinct symbols for all the major constructs in the schema except qualifiers (as opposed
5937 to properties, which are directly represented in the diagrams).

5938 In UML, a class is represented by a rectangle. The class name either stands alone in the rectangle or is in
5939 the uppermost segment of the rectangle. If present, the segment below the segment with the name
5940 contains the properties of the class. If present, a third region contains methods.

5941 A line decorated with a triangle indicates an inheritance relationship; the lower rectangle represents a
5942 subtype of the upper rectangle. The triangle points to the superclass.

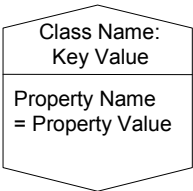
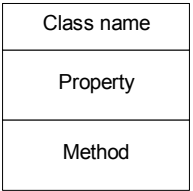
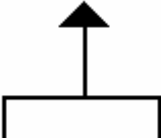
5943 Other solid lines represent relationships. The cardinality of the references on either side of the
5944 relationship is indicated by a decoration on either end. The following character combinations are
5945 commonly used:

- 5946 • "1" indicates a single-valued, required reference
- 5947 • "0..1" indicates an optional single-valued reference
- 5948 • "*" indicates an optional many-valued reference (as does "0..*")
- 5949 • "1..*" indicates a required many-valued reference

5950 A line connected to a rectangle by a dotted line represents a subclass relationship between two
5951 associations. The diagramming notation and its interpretation are summarized in Table D-1.

5952

Table D-1 – Diagramming Notation and Interpretation Summary

Meta Element	Interpretation	Diagramming Notation
Object		
Primitive type	Text to the right of the colon in the center portion of the class icon	
;Class		
Subclass		

Meta Element	Interpretation	Diagramming Notation
Association	1:1 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	
Trigger	State transition diagrams	
Qualifier	No direct equivalent	

ANNEX E (informative)

Guidelines

5953
5954
5955
5956

5957 The following are general guidelines for CIM modeling:

- 5958 • Method descriptions are recommended and must, at a minimum, indicate the method's side
5959 effects (pre- and post-conditions).
- 5960 • Leading underscores in identifiers are to be discouraged and not used at all in the standard
5961 schemas.
- 5962 • It is generally recommended that class names not be reused as part of property or method
5963 names. Property and method names are already unique within their defining class.
- 5964 • To enable information sharing among different CIM implementations, the MaxLen qualifier
5965 should be used to specify the maximum length of string properties.
- 5966 • When extending a schema (i.e., CIM schema or extension schema) with new classes, existing
5967 classes should be considered as superclasses of such new classes as appropriate, in order to
5968 increase schema consistency.

5969 E.1 SQL Reserved Words

5970 Avoid using SQL reserved words in class and property names. This restriction particularly applies to
5971 property names because class names are prefixed by the schema name, making a clash with a reserved
5972 word unlikely. The current set of SQL reserved words is as follows:

5973 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	

5974 From Annex E of sql1992.txt:

ABSOLUTE	ACTION	ADD	ALLOCATE
ALTER	ARE	ASSERTION	AT
BETWEEN	BIT	BIT_LENGTH	BOTH

CASCADE	CASCADED	CASE	CAST
CATALOG	CHAR_LENGTH	CHARACTER_LENGTH	COALESCE
COLLATE	COLLATION	COLUMN	CONNECT
CONNECTION	CONSTRAINT	CONSTRAINTS	CONVERT
CORRESPONDING	CROSS	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	DATE	DAY
DEALLOCATE	DEFERRABLE	DEFERRED	DESCRIBE
DESCRIPTOR	DIAGNOSTICS	DISCONNECT	DOMAIN
DROP	ELSE	END-EXEC	EXCEPT
EXCEPTION	EXECUTE	EXTERNAL	EXTRACT
FALSE	FIRST	FULL	GET
GLOBAL	HOUR	IDENTITY	IMMEDIATE
INITIALLY	INNER	INPUT	INSENSITIVE
INTERSECT	INTERVAL	ISOLATION	JOIN
LAST	LEADING	LEFT	LEVEL
LOCAL	LOWER	MATCH	MINUTE
MONTH	NAMES	NATIONAL	NATURAL
NCHAR	NEXT	NO	NULLIF
OCTET_LENGTH	ONLY	OUTER	OUTPUT
OVERLAPS	PAD	PARTIAL	POSITION
PREPARE	PRESERVE	PRIOR	READ
RELATIVE	RESTRICT	REVOKE	RIGHT
ROWS	SCROLL	SECOND	SESSION
SESSION_USER	SIZE	SPACE	SQLSTATE
SUBSTRING	SYSTEM_USER	TEMPORARY	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TRAILING	TRANSACTION	TRANSLATE	TRANSLATION
TRIM	TRUE	UNKNOWN	UPPER
USAGE	USING	VALUE	VARCHAR
VARYING	WHEN	WRITE	YEAR
ZONE			

5975 From Annex E of sql3part2.txt:

ACTION	ACTOR	AFTER	ALIAS
ASYNC	ATTRIBUTES	BEFORE	BOOLEAN
BREADTH	COMPLETION	CURRENT_PATH	CYCLE
DATA	DEPTH	DESTROY	DICTIONARY
EACH	ELEMENT	ELSEIF	EQUALS
FACTOR	GENERAL	HOLD	IGNORE
INSTEAD	LESS	LIMIT	LIST
MODIFY	NEW	NEW_TABLE	NO
NONE	OFF	OID	OLD
OLD_TABLE	OPERATION	OPERATOR	OPERATORS
PARAMETERS	PATH	PENDANT	POSTFIX
PREFIX	PREORDER	PRIVATE	PROTECTED
RECURSIVE	REFERENCING	REPLACE	ROLE
ROUTINE	ROW	SAVEPOINT	SEARCH
SENSITIVE	SEQUENCE	SESSION	SIMILAR
SPACE	SQLEXCEPTION	SQLWARNING	START
STATE	STRUCTURE	SYMBOL	TERM
TEST	THERE	TRIGGER	TYPE
UNDER	VARIABLE	VIRTUAL	VISIBLE

	WAIT	WITHOUT		
5976	From Annex E of sql3part4.txt:			
	CALL	DO	ELSEIF	EXCEPTION
	IF	LEAVE	LOOP	OTHERS
	RESIGNAL	RETURN	RETURNS	SIGNAL
	TUPLE	WHILE		

ANNEX F (normative)

EmbeddedObject and EmbeddedInstance Qualifiers

5981 Use of the EmbeddedObject and EmbeddedInstance qualifiers is motivated by the need to include the
5982 data of a specific instance in an indication (event notification) or to capture the contents of an instance at
5983 a point in time (for example, to include the CIM_DiagnosticSetting properties that dictate a particular
5984 CIM_DiagnosticResult in the Result object).

5985 Therefore, the next major version of the CIM Specification is expected to include a separate data type for
5986 directly representing instances (or snapshots of instances). Until then, the EmbeddedObject and
5987 EmbeddedInstance qualifiers can be used to achieve an approximately equivalent effect. They permit a
5988 CIM object manager (or other entity) to simulate embedded instances or classes by encoding them as
5989 strings when they are presented externally. Embedded instances can have properties that again are
5990 defined to contain embedded objects. CIM clients that do not handle embedded objects may treat
5991 properties with this qualifier just like any other string-valued property. CIM clients that do want to realize
5992 the capability of embedded objects can extract the embedded object information by decoding the
5993 presented string value.

5994 To reduce the parsing burden, the encoding that represents the embedded object in the string value
5995 depends on the protocol or representation used for transmitting the containing instance. This dependency
5996 makes the string value appear to vary according to the circumstances in which it is observed. This is an
5997 acknowledged weakness of using a qualifier instead of a new data type.

5998 This document defines the encoding of embedded objects for the MOF representation and for the CIM-
5999 XML protocol. When other protocols or representations are used to communicate with embedded object-
6000 aware consumers of CIM data, they must include particulars on the encoding for the values of string-
6001 typed elements qualified with EmbeddedObject or EmbeddedInstance.

6002 F.1 Encoding for MOF

6003 When the values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance are
6004 rendered in MOF, the embedded object must be encoded into string form using the MOF syntax for the
6005 instanceDeclaration nonterminal in embedded instances or for the classDeclaration,
6006 assocDeclaration, or indicDeclaration ABNF rules, as appropriate in embedded classes (see
6007 ANNEX A).

6008 EXAMPLES:

```
6009 instance of CIM_InstCreation {
6010     EventTime = "20000208165854.457000-360";
6011     SourceInstance =
6012         "instance of CIM_Fan {\n"
6013         "DeviceID = \"Fan 1\";\n"
6014         "Status = \"Degraded\";\n"
6015         "};\n";
6016 };
6017
6018 instance of CIM_ClassCreation {
6019     EventTime = "20031120165854.457000-360";
6020     ClassDefinition =
6021         "class CIM_Fan : CIM_CoolingDevice {\n"
```

```
6022     "    boolean VariableSpeed;\n"
6023         "        [Units (\\"Revolutions per Minute\\")]\n"
6024     "    uint64 DesiredSpeed;\n"
6025     "};\n"
6026 };
```

6027 **F.2 Encoding for CIM Protocols**

6028 The rendering of values of string-typed elements qualified with EmbeddedObject or EmbeddedInstance in
6029 CIM protocols is defined in the specifications defining these protocols.

ANNEX G (informative)

Schema Errata

6030
6031
6032
6033

6034 Based on the concepts and constructs in this document, the CIM schema is expected to evolve for the
6035 following reasons:

- 6036 • To add new classes, associations, qualifiers, properties and/or methods. This task is addressed
6037 in 5.3.
- 6038 • To correct errors in the Final Release versions of the schema. This task fixes errata in the CIM
6039 schemas after their final release.
- 6040 • To deprecate and update the model by labeling classes, associations, qualifiers, and so on as
6041 "not recommended for future development" and replacing them with new constructs. This task is
6042 addressed by the Deprecated qualifier described in 5.5.3.11.

6043 Examples of errata to correct in CIM schemas are as follows:

- 6044 • Incorrectly or incompletely defined keys (an array defined as a key property, or incompletely
6045 specified propagated keys)
- 6046 • Invalid subclassing, such as subclassing an optional association from a weak relationship (that
6047 is, a mandatory association), subclassing a nonassociation class from an association, or
6048 subclassing an association but having different reference names that result in three or more
6049 references on an association
- 6050 • Class references reversed as defined by an association's roles (antecedent/dependent
6051 references reversed)
- 6052 • Use of SQL reserved words as property names
- 6053 • Violation of semantics, such as missing Min(1) on a Weak relationship, contradicting that a
6054 weak relationship is mandatory

6055 Errata are a serious matter because the schema should be correct, but the needs of existing
6056 implementations must be taken into account. Therefore, the DMTF has defined the following process (in
6057 addition to the normal release process) with respect to any schema errata:

- 6058 a) Any error should promptly be reported to the Technical Committee (technical@dmf.org) for
6059 review. Suggestions for correcting the error should also be made, if possible.
- 6060 b) The Technical Committee documents its findings in an email message to the submitter within 21
6061 days. These findings report the Committee's decision about whether the submission is a valid
6062 erratum, the reasoning behind the decision, the recommended strategy to correct the error, and
6063 whether backward compatibility is possible.
- 6064 c) If the error is valid, an email message is sent (with the reply to the submitter) to all DMTF
6065 members (members@dmf.org). The message highlights the error, the findings of the Technical
6066 Committee, and the strategy to correct the error. In addition, the committee indicates the
6067 affected versions of the schema (that is, only the latest or all schemas after a specific version).
- 6068 d) All members are invited to respond to the Technical Committee within 30 days regarding the
6069 impact of the correction strategy on their implementations. The effects should be explained as
6070 thoroughly as possible, as well as alternate strategies to correct the error.

- 6071 e) If one or more members are affected, then the Technical Committee evaluates all proposed
6072 alternate correction strategies. It chooses one of the following three options:
- 6073 – To stay with the correction strategy proposed in b)
 - 6074 – To move to one of the proposed alternate strategies
 - 6075 – To define a new correction strategy based on the evaluation of member impacts
- 6076 f) If an alternate strategy is proposed in Item e), the Technical Committee may decide to reenter
6077 the errata process, resuming with Item c) and send an email message to all DMTF members
6078 about the alternate correction strategy. However, if the Technical Committee believes that
6079 further comment will not raise any new issues, then the outcome of Item e) is declared to be
6080 final.
- 6081 g) If a final strategy is decided, this strategy is implemented through a Change Request to the
6082 affected schema(s). The Technical Committee writes and issues the Change Request. Affected
6083 models and MOF are updated, and their introductory comment section is flagged to indicate that
6084 a correction has been applied.

ANNEX H (informative)

Ambiguous Property and Method Names

6089 In 5.1.2.8 it is explicitly allowed for a subclass to define a property that may have the same name as a
6090 property defined by a superclass and for that new property not to override the superclass property. The
6091 subclass may override the superclass property by attaching an Override qualifier; this situation is well-
6092 behaved and is not part of the problem under discussion.

6093 Similarly, a subclass may define a method with the same name as a method defined by a superclass
6094 without overriding the superclass method. This annex refers only to properties, but it is to be understood
6095 that the issues regarding methods are essentially the same. For any statement about properties, a similar
6096 statement about methods can be inferred.

6097 This same-name capability allows one group (the DMTF, in particular) to enhance or extend the
6098 superclass in a minor schema change without to coordinate with, or even to know about, the development
6099 of the subclass in another schema by another group. That is, a subclass defined in one version of the
6100 superclass should not become invalid if a subsequent version of the superclass introduces a new
6101 property with the same name as a property defined on the subclass. Any other use of the same-name
6102 capability is strongly discouraged, and additional constraints on allowable cases may well be added in
6103 future versions of CIM.

6104 It is natural for CIM clients to be written under the assumption that property names alone suffice to
6105 identify properties uniquely. However, such CIM clients risk failure if they refer to properties from a
6106 subclass whose superclass has been modified to include a new property with the same name as a
6107 previously-existing property defined by the subclass.

6108 For example, consider the following:

```
6109 [Abstract]
6110 class CIM_Superclass
6111 {
6112 };
6113
6114 class VENDOR_Subclass
6115 {
6116     string Foo;
6117 };
```

6118 Assuming CIM-XML as the CIM protocol and assuming only one instance of VENDOR_Subclass,
6119 invoking the EnumerateInstances operation on the class "VENDOR_Subclass" without also asking for
6120 class origin information might produce the following result:

```
6121 <INSTANCE CLASSNAME="VENDOR_Subclass">
6122     <PROPERTY NAME="Foo" TYPE="string">
6123         <VALUE>Hello, my name is Foo</VALUE>
6124     </PROPERTY>
6125 </INSTANCE>
```

6126 If the definition of CIM_Superclass changes to:

```
6127 [Abstract]
```

```

6128 class CIM_Superclass
6129 {
6130     string Foo = "You lose!";
6131 };

```

6132 then the EnumerateInstances operation might return the following:

```

6133 <INSTANCE>
6134     <PROPERTY NAME="Foo" TYPE="string">
6135         <VALUE>You lose!</VALUE>
6136     </PROPERTY>
6137     <PROPERTY NAME="Foo" TYPE="string">
6138         <VALUE>Hello, my name is Foo</VALUE>
6139     </PROPERTY>
6140 </INSTANCE>

```

6141 If the CIM client attempts to retrieve the 'Foo' property, the value it obtains (if it does not experience an
6142 error) depends on the implementation.

6143 Although a class may define a property with the same name as an inherited property, it may not define
6144 two (or more) properties with the same name. Therefore, the combination of defining class plus property
6145 name uniquely identifies a property. (Most CIM operations that return instances have a flag controlling
6146 whether to include the class origin for each property. For example, in DSP0200, see the clause on
6147 EnumerateInstances; in DSP0201, see the clause on ClassOrigin.)

6148 However, the use of class-plus-property-name for identifying properties makes a CIM client vulnerable to
6149 failure if a property is promoted to a superclass in a subsequent schema release. For example, consider
6150 the following:

```

6151 class CIM_Top
6152 {
6153 };
6154
6155 class CIM_Middle : CIM_Top
6156 {
6157     uint32 Foo;
6158 };
6159
6160 class VENDOR_Bottom : CIM_Middle
6161 {
6162     string Foo;
6163 };

```

6164 A CIM client that identifies the uint32 property as "the property named 'Foo' defined by CIM_Middle" no
6165 longer works if a subsequent release of the CIM schema changes the hierarchy as follows:

```

6166 class CIM_Top
6167 {
6168     uint32 Foo;
6169 };
6170
6171 class CIM_Middle : CIM_Top
6172 {
6173 };

```

```
6174
6175 class VENDOR_Bottom : CIM_Middle
6176 {
6177     string Foo;
6178 };
```

6179 Strictly speaking, there is no longer a "property named 'Foo' defined by CIM_Middle"; it is now defined by
6180 CIM_Top and merely inherited by CIM_Middle, just as it is inherited by VENDOR_Bottom. An instance of
6181 VENDOR_Bottom returned in CIM-XML from a CIM server might look like this:

```
6182 <INSTANCE CLASSNAME="VENDOR_Bottom">
6183     <PROPERTY NAME="Foo" TYPE="string" CLASSORIGIN="VENDOR_Bottom">
6184         <VALUE>Hello, my name is Foo!</VALUE>
6185     </PROPERTY>
6186     <PROPERTY NAME="Foo" TYPE="uint32" CLASSORIGIN="CIM_Top">
6187         <VALUE>47</VALUE>
6188     </PROPERTY>
6189 </INSTANCE>
```

6190 A CIM client looking for a PROPERTY element with NAME="Foo" and CLASSORIGIN="CIM_Middle" fails
6191 with this XML fragment.

6192 Although CIM_Middle no longer defines a 'Foo' property directly in this example, we intuit that we should
6193 be able to point to the CIM_Middle class and locate the 'Foo' property that is defined in its nearest
6194 superclass. Generally, a CIM client must be prepared to perform this search, separately obtaining
6195 information, when necessary, about the (current) class hierarchy and implementing an algorithm to select
6196 the appropriate property information from the instance information returned from a CIM operation.

6197 Although it is technically allowed, schema writers should not introduce properties that cause name
6198 collisions within the schema, and they are strongly discouraged from introducing properties with names
6199 known to conflict with property names of any subclass or superclass in another schema.

ANNEX I (informative)

OCL Considerations

6200
6201
6202
6203

6204 The Object Constraint Language (OCL) is a formal language to describe expressions on models. It is
6205 defined by the Open Management Group (OMG) in the [Object Constraint Language](#) specification, which
6206 describes OCL as follows:

6207 OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side
6208 effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the
6209 model. This means that the state of the system will never change because of the evaluation of an OCL
6210 expression, even though an OCL expression can be used to specify a state change (e.g., in a post-
6211 condition).

6212 OCL is not a programming language; therefore, it is not possible to write program logic or flow control in
6213 OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a
6214 modeling language in the first place, OCL expressions are not by definition directly executable.

6215 OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL
6216 expression must conform to the type conformance rules of the language. For example, you cannot
6217 compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL
6218 type. In addition, OCL includes a set of supplementary predefined types. These are described in Chapter
6219 11 ("The OCL Standard Library").

6220 As a specification language, all implementation issues are out of scope and cannot be expressed in OCL.
6221 The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model
6222 cannot change during evaluation."

6223 For a particular CIM class, more than one CIM association referencing that class with one reference can
6224 define the same name for the opposite reference. OCL allows navigation from an instance of such a class
6225 to the instances at the other end of an association using the name of the opposite association end (that
6226 is, a CIM reference). However, in the case discussed, that name is not unique. For OCL statements to
6227 tolerate the future addition of associations that create such ambiguity, OCL navigation from an instance to
6228 any associated instances should first navigate to the association class and from there to the associated
6229 class, as described in the [Object Constraint Language](#) specification in its sections 7.5.4 "Navigation to
6230 Association Classes" and 7.5.5 "Navigation from Association Classes". OCL requires the first letter of the
6231 association class name to be lowercase when used for navigating to it. For example, CIM_Dependency
6232 becomes cim_Dependency.

6233 EXAMPLE:

```
6234 [ClassConstraint {
6235     "inv i1: self.p1 = self.acme_A12.r.p2"}]
6236     // Using class name ACME_A12 is required to disambiguate end name r
6237 class ACME_C1 {
6238     string p1;
6239 };
6240
6241 [ClassConstraint {
6242     "inv i2: self.p2 = self.acme_A12.x.p1", // Using ACME_A12 is recommended
6243     "inv i3: self.p2 = self.x.p1"}] // Works, but not recommended
6244 class ACME_C2 {
6245     string p2;
```

```
6246 };
6247
6248 class ACME_C3 { };
6249
6250     [Association]
6251 class ACME_A12 {
6252     ACME_C1 REF x;
6253     ACME_C2 REF r; // same name as ACME_A13::r
6254 };
6255
6256     [Association]
6257 class ACME_A13 {
6258     ACME_C1 REF y;
6259     ACME_C3 REF r; // same name as ACME_A12::r
6260 };
```

ANNEX J (informative)

Change Log

6261
6262
6263
6264

Version	Date	Description
1	1997-04-09	First Public Release
2.2	1999-06-14	Released as Final Standard
2.2.1000	2003-06-07	Released as Final Standard
2.3	2004-08-11	Released as Preliminary Standard
2.3	2005-10-04	Released as Final Standard
2.4.0a	2007-11-12	Released as Preliminary Standard <ul style="list-style-type: none"> • ARCHCR00038.009 - Add Correlatable qualifier • ARCHCR00039.011 - Datetime Arithmetic and Comparison • ARCHCR00050.003 - Datetime Calendar 2 • ARCHCR00050.006 - Datetime Calendar 2 • ARCHCR00056.004 - Datetime Special Values • ARCHCR00057.003 - OCL Qualifier Extension • ARCHCR00067.001 - Clarify that qualifiers do not adorn qualifier types, or other qualifiers • ARCHCR00068.001 - Clarifications for overriding qualifiers • ARCHCR00069.003 - Clarify qualifiers MinValue,MaxValue,MinLen,MaxLen • ARCHCR00070.003 - Clarifications on KEY qualifier (including NULL) • ARCHCR00071.005 - Add new DisplayDescription qualifier • ARCHCR00072.014 - Deprecate units and add programmatic units • ARCHCR00073.000 - real32+64 are IEEE-754 • ARCHCR00074.004 - Resolve inconsistency in scopes of REQUIRED qualifier • ARCHCR00075.001 - Add method scope to REQUIRED qualifier • ARCHCR00076.002 - Clarify format of OVERRIDE qualifier including NULL • ARCHCR00077.000 - Add default rule for UmlPackagePath qualifier • ARCHCR00078.000 - Minor cleanup re qualifier duplication • ARCHCR00079.000 - Minor cleanup in section 2.5.1 • ARCHCR00080.000 - Minor cleanup in section 4.7 • ARCHCR00081.009 - Various clarifications on arrays • ARCHCR00082.003 - Clarification of static concept • ARCHCR00083.000 - KEY properties cannot be embedded objects • ARCHCR00085.000 - Clarify NULL for Min and Max qualifiers • ARCHCR00090.003 - Clarify EXPERIMENTAL qualifier • ARCHCR00091.007 - Programmatic units for counted phenomenons • ARCHCR00092.001 - Mandate schema name to be first segment of UMLPackagePath • ARCHCR00093.004 - Several clarifications for Deprecated qualifier • ARCHCR00094.004 - Clarifications for syntax of MappingString qualifier • ARCHCR00097.004 - Clarify the target of the Override qualifier • ARCHCR00098.005 - Clarify OCL related qualifiers • ARCHCR00099.004 - Clarify EmbeddedInstance and EmbeddedObject qualifiers • ARCHCR00101.004 - Clarify any datatype may be NULL and clarify NULL-ness of references in associations • ARCHCR00106.001 - Fix KEY qualifier definition
2.5.0a	2008-04-22	Released as Preliminary Standard <ul style="list-style-type: none"> • various formal changes to follow ISO Guidelines

Version	Date	Description
2.5.0	2009-03-04	Released as DMTF Standard, with the following changes: <ul style="list-style-type: none"> • ARCHCR00129.001 - Reduce programmatic units for counted items to just one • ARCHCR00130.001 - Fix name of programmatic unit for rack unit
2.6.0a	2009-11-04	Released as a Work in Progress, with the following changes: <ul style="list-style-type: none"> • ARCHCR00103.007 - Prohibit some DisableOverride qualifiers on overriding elements • ARCHCR00107.004 - BNF defined format for datetime values • ARCHCR00109.006 - Clarify usage of programmatic units by datatype • ARCHCR00110.009 - Glossary clarifications and additions • ARCHCR00112.005 - Clarify OctetString qualifier • ARCHCR00116.010 - Renovation of object naming and initializers • ARCHCR00117.004 - Comparison of values • ARCHCR00118.003 - Clarify schema and qualifier type modifications • ARCHCR00119.001 - Add XML strings via the XMLNamespaceName qualifier • ARCHCR00120.002 - Qualifier type declaration for XMLNamespaceName • ARCHCR00121.000 - Clarify Propagated and Weak qualifiers • ARCHCR00122.001 - Move normative text out of informative Guidelines annex • ARCHCR00123.002 - Renovate CIM Meta Schema • ARCHCR00126.003 - Clarify qualifier concept including flavors • ARCHCR00128.002 - Deprecate the Translatable qualifier flavor • ARCHCR00132.000 - Add programmatic units VA and VAR • ARCHCR00133.002 - Clarifications for property default value • ARCHCR00134.002 - Fix compatibility statement for Required qualifier • ARCHCR00135.000 - Fix several flaws in ValueMap description • ARCHCR00136.002 - Clarify extensibility of string based ValueMap • ARCHCR00137.000 - Deprecate implicit qualifiers and element level flavors • ARCHCR00138.002 - Deprecate covered elements in the same schema • ARCHCR00139.000 - Define qualifier flavors and other qualifier clarifications • ARCHCR00140.001 - Add Value class to meta schema + other fixes • ARCHCR00141.003 - Clarify Unicode support and deprecate the char16 type
2.6.0	2010-03-17	Released as DMTF Standard, with the following changes: <ul style="list-style-type: none"> • Made the IsPUnit qualifier final by removing its "experimental" status. • Added or changed terms: CIM server, CIM client, CIM operation, CIM protocol, CIM listener, implicit qualifier, references to document related terms in ISO guidelines. • Corrected errors (relative to the qualifiers.mof and qualifiers_optional.mof files in the CIM Schema) of data type, scope, flavor and/or default value of the following qualifiers: Indication, ArrayType, Counter, Description, DN, EmbeddedInstance, IsPUnit, Key, MaxValue, Revision, Expensive, UnknownValues, UnsupportedValues. • Corrected that the ABNF rules defining the General Mapping String Format for the MappingStrings qualifier are required to be assembled without intervening whitespace. This was (incorrectly) only a recommendation before. • Corrected the specification of parameter names in ModelCorrespondence qualifier. • Fixed incorrect "64-bit" in description of length in OctetString qualifier to become "32-bit". • Clarified the implications of the requirements on the UCS/Unicode character repertoire for CIM clients that still use UCS-2. • Extended scope of Annex F (Embedded Objects) to cover all CIM protocols, and removed information that belongs into CIM protocol specifications. • Moved some CIM protocol related references from Normative References to Bibliography.

Bibliography

6265

- 6266 Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Document Set*,
6267 Rational Software Corporation, 1996, <http://www.rational.com/uml>
- 6268 James O. Coplein, Douglas C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley,
6269 Reading Mass., 1995
- 6270 Georges Gardarin and Patrick Valduriez, *Relational Databases and Knowledge Bases*, Addison Wesley,
6271 1989
- 6272 Gerald M. Weinberg, *An Introduction to General Systems Thinking*, 1975 ed. Wiley-Interscience, 2001 ed.
6273 Dorset House
- 6274 DMTF DSP0200, *CIM Operations over HTTP*, Version 1.3
6275 http://www.dmtf.org/standards/published_documents/DSP0200_1.3.pdf
- 6276 DMTF DSP0201, *Specification for the Representation of CIM in XML*, Version 2.3
6277 http://www.dmtf.org/standards/published_documents/DSP0201_2.3.pdf
- 6278 ISO/IEC 19757-2:2008, *Information technology -- Document Schema Definition Language (DSDL) -- Part*
6279 *2: Regular-grammar-based validation -- RELAX NG*,
6280 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52348
- 6281 IETF, RFC2068, *Hypertext Transfer Protocol – HTTP/1.1*, <http://tools.ietf.org/html/rfc2068>
- 6282 IETF, RFC1155, *Structure and Identification of Management Information for TCP/IP-based Internets*,
6283 <http://tools.ietf.org/html/rfc1155>
- 6284 IETF, RFC2253, *Lightweight Directory Access Protocol (v3): UTF-8 String Representation Of*
6285 *Distinguished Names*, <http://tools.ietf.org/html/rfc2253>
- 6286 OMG, *Unified Modeling Language: Infrastructure*, Version 2.1.1
6287 <http://www.omg.org/cgi-bin/doc?formal/07-02-06>
- 6288 The Unicode Consortium: *The Unicode Standard*, <http://www.unicode.org>
- 6289 W3C, *Character Model for the World Wide Web 1.0: Normalization*, Working Draft, 27 October 2005,
6290 <http://www.w3.org/TR/charmod-norm/>
- 6291 W3C, *XML Schema Part 0: Primer Second Edition*, W3C Recommendation, 28 October 2004,
6292 <http://www.w3.org/TR/xmlschema-0/>