



1
2
3
4
5

Document Number: DSP0004

Date: 2011-10-18

Version: 3.0.0a

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

Information for Work-in-Progress version:

This document is subject to change at any time without further notice.

It expires on: 04/18/2012

Provide any comments through the DMTF Feedback Portal:

<http://www.dmtf.org/standards/feedback>

IMPORTANT: This specification is not a standard. It does not necessarily reflect the views of the DMTF or all of its members. Because this document is a Work in Progress, this specification may still change, perhaps profoundly. This document is available for public review and comment until the stated expiration date.

7

- 8 **Document Type: Specification**
- 9 **Document Status: Work in Progress Specification - not a DMTF Standard**
- 10 **Document Language: US-EN**

Copyright Notice

Copyright © 2011 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

11

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26 withdrawn or modified after publication, and shall be indemnified and held harmless by any party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

33

CONTENTS

35	Foreword	7
36	Document Conventions	7
37	Introduction	9
38	1 Scope	11
39	2 Normative references	11
40	3 Terms and definitions	13
41	4 Symbols and abbreviated terms	16
42	5 Formal syntax considerations	16
43	5.1 Specification of attributes	16
44	5.2 Specification of associations	16
45	5.3 Specification of constraints	17
46	6 CIM metamodel	17
47	6.1 Introduction	17
48	6.2 Concepts	18
49	6.2.1 Inheritance	18
50	6.2.2 Multiplicity	18
51	6.2.3 Association	18
52	6.3 Metamodel overview	20
53	6.4 Principal metaelements	21
54	6.4.1 CIMM::Association	21
55	6.4.2 CIMM::Class	22
56	6.4.3 CIMM::Element	23
57	6.4.4 CIMM::Enumeration	23
58	6.4.5 CIMM::EnumerationLiteral	24
59	6.4.6 CIMM::Instance	24
60	6.4.7 CIMM::InstanceValue	25
61	6.4.8 CIMM::LiteralSpecification	25
62	6.4.9 CIMM::Method	26
63	6.4.10 CIMM::MethodReturn	27
64	6.4.11 CIMM::NamedElement	28
65	6.4.12 CIMM::NamingContext	30
66	6.4.13 CIMM::OpaqueExpression	30
67	6.4.14 CIMM::Parameter	31
68	6.4.15 CIMM::PrimitiveType	32
69	6.4.16 CIMM::Property	36
70	6.4.17 CIMM::Qualifier	37
71	6.4.18 CIMM::QualifierType	39
72	6.4.19 CIMM::Schema	40
73	6.4.20 CIMM::Slot	41
74	6.4.21 CIMM::Structure	41
75	6.4.22 CIMM::Type	42
76	6.4.23 CIMM::TypedElement	43
77	6.4.24 CIMM::ValueSpecification	44
78	7 QualifierTypes	47
79	7.1 Core metamodel extensions	47
80	7.1.1 AggregationKind	47
81	7.1.2 ArrayType	48
82	7.1.3 Description	48
83	7.1.4 Max	48
84	7.1.5 MaxLen	49
85	7.1.6 MaxValue	49

86	7.1.7	Min	50
87	7.1.8	MinLen	50
88	7.1.9	MinValue	50
89	7.1.10	NullOK (proposed)	51
90	7.1.11	Override	51
91	7.1.12	SchemaURI (proposed)	52
92	7.1.13	Read	52
93	7.1.14	Write	52
94	7.1.15	Version	53
95	7.2	Specification based metamodel extensions	53
96	7.2.1	Correlatable	53
97	7.2.2	MappingStrings	54
98	7.2.3	ModelCorrespondence	54
99	7.2.4	OCLConstraint	56
100	7.3	Descriptive metamodel extensions	57
101	7.3.1	Counter	57
102	7.3.2	Deprecated	57
103	7.3.3	DisplayName	58
104	7.3.4	Experimental	58
105	7.3.5	Guage	59
106	7.3.6	IsPUnit	59
107	7.3.7	ResourceDescription (proposed)	60
108	7.3.8	PUnit	60
109	7.3.9	XMLNamespaceName	60
110	8	Model element naming	62
111	8.1	Model element name syntax	63
112	8.2	Matching model element names	63
113	8.3	Identity of CIM objects	64
114	9	Supported schema modifications	64
115	10	Object constraint language	70
116	10.1	Navigation across associations	71
117	10.2	Self	71
118	10.3	Types	71
119	10.4	Operations and precedence	71
120	10.4.1	OCL expression keywords	72
121	10.4.2	OCL operations	72
122	10.5	OCL statements	73
123	10.5.1	Comment statement	73
124	10.5.2	Let expressions	73
125	10.5.3	OCL definition constraints	74
126	10.5.4	OCL invariant constraints	74
127	10.5.5	OCL precondition constraint	74
128	10.5.6	OCL postcondition constraint	75
129	10.5.7	OCL body constraint	75
130	10.5.8	OCL derivation constraint	75
131	10.5.9	OCL initialization constraint	75
132	10.6	OCL constraint examples	76
133	Annex A	(informative) Type lattice	78
134	Annex B	(normative) DateTime	79
135	Annex C	(normative) Backwards compatability rules for schema modifications	82
136	Annex D	(normative) UCS and Unicode	84
137	Annex E	(normative) Comparison of values	85
138	Annex F	(normative) Programmatic units	87
139	Annex G	(normative) MappingStrings formats	93

140	Annex H (informative) Modeling guidelines	96
141	Annex I (informative) Change log	99
142	Bibliography	100
143		
144	Figures	
145	Figure 1: Overview CIM Metamodel Diagram	20
146	Figure 2: WBEM Infrastructure.....	62
147		
148	Tables	
149	Table 1: Standards Bodies.....	11
150	Table 2: Specializations of LiteralSpecification.....	26
151	Table 3: Predefined Types	32
152	Table 4: Specializations of ValueSpecification	44
153	Table 5: Compatibility of Schema Modifications	65
154	Table 6: Compatibility of Qualifier Type Modifications	69
155	Table 7: Operations.....	71
156	Table 8: OCL Expression keywords.....	72
157	Table 9: OCL Operations	72
158	Table 10: Evaluation of DateTime Expressions	81
159	Table 11: Changes that Increment the CIM Schema Major Version	82
160	Table 12: Base Units for Programmatic Units.....	89
161	Table 13: Example MappingStrings mapping	95
162		

163

Foreword

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

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

168 Acknowledgments

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

170 Editor:

- 171 • George Ericson – EMC

172 Contributors:

- 173 • Jim Davis – WBEM Solutions
- 174 • Wojtek Kozaczynski – Microsoft
- 175 • Lawrence Lamers – VMware
- 176 • Andreas Maier – IBM
- 177 • Karl Schopmeyer – Inova Development

178 Document Conventions

179 Typographical Conventions

180 The following typographical conventions are used in this document:

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

184 ABNF Usage Conventions

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

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

192 Deprecated Material

193 Deprecated material is not recommended for use in new development efforts. Existing and new clients
194 may rely this material, but should move to the favored approach as soon as possible. Conformant
195 implementations shall implement any deprecated elements as required by this document in order to
196 achieve backwards compatibility.

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

199 The following typographical convention indicates deprecated material:

200 **DEPRECATED**

201 Deprecated material appears here.

202 **DEPRECATED**

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

205 **Experimental Material**

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

211 The following typographical convention indicates experimental material:

212 **EXPERIMENTAL**

213 Experimental material appears here.

214 **EXPERIMENTAL**

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

217

Introduction

218 This document specifies the DMTF Common Information Model Metamodel (CIMM). The role of CIMM is
219 to define the semantics for the construction of conformant models. The Common Information Model
220 (CIM) is an example of a set of models that are conformant with the CIMM. The CIM is composed of a
221 core model and a number of common models that define a rich and detailed ontology for computer and
222 systems management.

223 Each model may be represented by one or more schemas. The classes defined by a CIMM conformant
224 schema include methods as well as properties and so enable both active management and
225 instrumentation.

226 One benefit of CIMM is that conformant models may extend, or intermix with other conformant models.
227 Vendors or others outside of the DMTF may also create CIMM conformant models. Such models are not
228 required to extend existing CIM models, but may do so.

229 The CIM metamodel is based on a subset of the UML metamodel, (as defined in OMG, Unified Modeling
230 Language: Superstructure), with the intention that most existing schemas can interoperate with, (or be
231 incorporated into), the Common Information Model with little or no modification.

232 Derivation from UML additionally enables the use of commonly available UML tools to create and manage
233 CIM schemas and related artifacts.

234 The Unified Modeling Language (UML) from the Object Management Group (OMG) allows users to
235 specify systems. UML includes twelve diagram types to allow various aspects of a system's design to be
236 visualized. These diagrams are:

- 237 • **Structural Diagrams** include the Structure Diagram, Composite Structure Diagram,
238 Component Diagram, Deployment Diagram, Object Diagram, Schema Diagram, and Profile
239 Diagram
- 240 • **Behavior Diagrams** include the Activity Diagram, Communication Diagram, Interaction
241 Overview Diagram, Sequence Diagram, State Machine Diagram, Timing Diagram, and the Use
242 Case Diagram

243 One of the big advantages of using UML tools is that they provide a wealth of capabilities beyond just
244 representing CIM models. For example, they can be used to describe state diagrams, use cases or
245 interactions between management applications and managed elements.

246 Common Information Model (CIM) Metamodel

247 1 Scope

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

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

253 2 Normative references

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

258 Table 1 shows standards bodies and their web sites.

259 **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
OMG	Open Management Group	http://www.omg.org
W3C	World Wide Web Consortium	http://www.w3.org

260 ANSI/IEEE 754-1985, IEEE® Standard for Floating-Point Arithmetic, August 29 1985
 261 <http://ieeexplore.ieee.org/servlet/opac?punumber=4610933>

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

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

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

268 IEEE Std 1003.1, 2004 Edition, Standard for information technology - portable operating system interface
269 (POSIX). Shell and utilities
270 http://www.unix.org/version3/ieee_std.html

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

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

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

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

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

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

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

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

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

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

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

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

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

304 ISO/IEC 10646:2003, Information technology — Universal Multiple-Octet Coded Character Set (UCS)
305 [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)

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

- 310 ISO/IEC 10646:2003/Amd 2:2006, Information technology — Universal Multiple-Octet Coded Character
311 Set (UCS) — Amendment 2: N'Ko, Phags-pa, Phoenician and other characters
312 [http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006\(E\).
313 zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/c041419_ISO_IEC_10646_2003_Amd_2_2006(E).zip)
- 314 ISO/IEC 14651:2007, Information technology — International string ordering and comparison — Method
315 for comparing character strings and description of the common template tailorable ordering
316 [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)
- 317 ISO/IEC 14750:1999, Information technology — Open Distributed Processing — Interface Definition
318 Language
319 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=25486
- 320 ITU X.501, Information Technology — Open Systems Interconnection — The Directory: Models
321 <http://www.itu.int/rec/T-REC-X.501/en>
- 322 ITU X.680 (07/02), Information technology — Abstract Syntax Notation One (ASN.1): Specification of
323 basic notation
324 <http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf>
- 325 OMG, Object Constraint Language, Version 2.2
326 <http://www.omg.org/spec/OCL/2.2>
- 327 OMG, Unified Modeling Language: Infrastructure, Version 2.3
328 <http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/>
- 329 OMG, Unified Modeling Language: Superstructure, Version 2.3
330 <http://www.omg.org/spec/UML/2.3/Superstructure/PDF/>
- 331 The Unicode Consortium, The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization
332 Forms
333 <http://www.unicode.org/reports/tr15/>
- 334 W3C, NamingContexts in XML, W3C Recommendation, 14 January 1999
335 <http://www.w3.org/TR/REC-xml-names>

336 **3 Terms and definitions**

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

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

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

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

350 The following additional terms are used in this document.

351 **3.1**

- 352 **Cardinality**
353 the number of elements
- 354 **3.2**
355 **Common Information Model Metamodel**
356 **CIM Metamodel**
357 **CIMM**
358 **Error! Reference source not found.** used in the realization of CIM elements
- 359 **3.3**
360 **CIMM schema**
361 a schema, (including but not limited to CIM schema), that is CIMM conformant
- 362 **3.4**
363 **CIM schema**
364 CIMM conformant schema published by the DMTF that represents the Common Information Model
- 365 **3.5**
366 **Common Information Model**
367 **CIM**
368 a set of management models developed by the DMTF for the management of computing systems. The
369 set is composed of a core model and a set of common models. Each model is CIM metamodel
370 conformant schema published by the DMTF (i.e., the CIM schema).
- 371 **3.6**
372 **key (of an instance)**
373 a unique identifier for the instance within a WBEM service.
- 374 **3.7**
375 **key property**
376 property of a class, with a value that is a unique identifier for an instance of that class
- 377 **3.8**
378 **metamodel**
379 model that defines the semantics and rules of the elements and relationships used in the realization of
380 conformant models
381 For example, the CIM metamodel includes the includes a definition of "Structure" that allows "Properties"
382 and not "Methods." As a result, a CIM metamodel conformant model that defines a "Structure" S1 may
383 include "Properties" P1 and P2, but may not define "Methods" on S1.
- 384 **3.9**
385 **metaschema**
386 schema that represents a metamodel
- 387 **3.10**
388 **model**
389 set of conceptual elements and the relationships between them that collectively define the semantics,
390 behavior and state of some thing
- 391 **3.11**

392 Managed Object Format**393 MOF**

394 A language that describes CIMM schema
395 elements in textual form.

396 3.12**397 multiplicity**

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

401 3.13**402 schema**

403 a formal language representation of a model. For example a MOF representation of the CIM model
404 defines a CIM schema

405 3.14**406 WBEM client**

407 An entity responsible for originating WBEM operations for processing by a WBEM service.
408 This definition does not imply any particular implementation architecture or scope, such as a client library
409 component or an entire management application.

410 3.15**411 WBEM indication**

412 an interaction originated on a WBEM service and processed by a WBEM listener.

413 3.16**414 WBEM indication confirmation**

415 an interaction originated on a WBEM listener in response to a WBEM indication and processed by a
416 WBEM service.

417 3.17**418 WBEM listener**

419 an entity responsible for processing WBEM indications originated by a WBEM service.
420 This definition does not imply any particular implementation architecture or scope, such as a standalone
421 demon component or an entire management application.

422 3.18**423 WBEM operation**

424 a request originated by a WBEM client and processed by a WBEM service.

425 3.19**426 WBEM operation reply**

427 a request originated by a WBEM service in response to a WBEM operation and processed by a WBEM
428 client.

429 3.20**430 WBEM protocol**

431 a protocol used to pass WBEM operation, WBEM indication, WBEM operation reply, and WBEM
432 indication confirmation messages between WBEM client, WBEM service and WBEM listener entities.
433 This definition does not imply any particular communication protocol stack, or even that the protocol
434 performs a remote communication.

435 3.21

436 **WBEM service**
 437 an entity responsible for processing WBEM operation requests and WBEM indication confirmation
 438 messages, and for originating WBEM indication and WBEM operation reply messages.
 439

440 **4 Symbols and abbreviated terms**

441 The following abbreviations are used in this document.

442 **MIB**

443 Management Information Base

444 **MIF**

445 Management Information Format

446 **OID**

447 object identifier

448 **SNMP**

449 Simple Network Management Protocol

450 **UML**

451 Unified Modeling Language

452 **5 Formal syntax considerations**

453 **5.1 Specification of attributes**

454 Descriptions of attributes use the attribute-format ABNF rule (whitespace allowed):

```

455 attribute-format =
456   attr-name ":" attr-type ("[" attr-multiplicity "]" ("{"qualifier-list"}")
457   ; the format used to describe the attributes of CIM Metaelements
458
459 attr-name = IDENTIFIER
460   ; the name of the attribute
461
462 attr-type = type
463   ; the datatype of the attribute
464
465 type = "string" ; a string of UCS characters of arbitrary length
466   / "boolean" ; a boolean value
467   / "integer" ; a signed 64-bit integer value
468
469 attr-multiplicity = cardinality-format
470   ; the multiplicity of the attribute. The default multiplicity is 1
  
```

471 **5.2 Specification of associations**

472 Descriptions of association ends use the association-end-format ABNF rule (whitespace allowed):

```

473 association-end-format = other-role ":" other-element "[" other-cardinality "]"
474   ; the format used to describe association ends of associations
  
```

```

475 ; between CIM Metaelements
476
477 other-role = IDENTIFIER
478 ; the role of the association end (on this side of the relationship)
479 ; that is referencing the associated metaelement
480
481 other-element = IDENTIFIER
482 ; the name of the associated metaelement
483
484 other-cardinality = cardinality-format
485 ; the cardinality of the associated metaelement
486
487 cardinality-format = positiveIntegerValue ; exactly that
488 / "*" ; zero to any
489 / integerValue ".." positiveIntegerValue ; min to max
490
491 / integerValue ".." "*" ; min to any
492 ; format of a cardinality specification
493
494 integerValue = decimalDigit *decimalDigit ; no whitespace allowed
495
496 positiveIntegerValue = positiveDecimalDigit *decimalDigit ; no whitespace allowed

```

497 5.3 Specification of constraints

498 This specification defines constraints that shall be met by implementations that are conformant to the CIM
 499 metamodel. These constraints fall into two categories:

- 500 • OCL constraints – Constraints defined using a subset of the Object Constraint Language (OCL)
 501 as defined in clause 10. This is the main category of constraints.
- 502 • Other constraints – Constraints defined using normative text. This category only exists for
 503 constraints for which it was not possible to define an according OCL statement.

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

506 6 CIM metamodel

507 6.1 Introduction

508 The CIM metamodel provides the basis on which CIM schemas and models are defined.

509 The CIM metamodel is based on a subset of elements from the Unified Modeling Language (UML), see
 510 OMG, Unified Modeling Language: Superstructure .

511 This document assumes familiarity with UML notation and with basic object-oriented concepts in the form
 512 of classes, properties, methods, operations, inheritance, associations, objects, multiplicity, and
 513 polymorphism.

514 To precisely specify constraints, a subset of the OMG [Object Constraint Language](#) (OCL) as defined in
 515 clause 10.

516 The metamodel defines the structure and behavior of modeled elements that are realized from
517 metamodel elements.

518 The metamodel consists of metaelements that have attributes, operations, and associations with other
519 metaelements. For example, a class in a model is realized from a class metaelement that has attributes
520 such as a class name, and relationships such as a generalization relationship to a superclass, or
521 ownership relationships to its properties and methods.

522 Note: To help distinguish between characteristics of the metaelements of a metamodel and those of the
523 elements of a model, the terms attribute and operation are used when referencing a metaelement in the
524 metamodel and the corresponding terms property and method are used when referencing an element in the
525 model.

526 Each metaelement represents semantic aspects of model elements realized from the metaelement or a
527 related metaelement.

528 The attribute values allow the state of the metaelement to be examined and governed. Operations
529 provide a means to examine or modify the behavior and state of the metaelement.

530 Note: Metamodel elements affect the behavior and structure of model elements and not the modeled element.
531 For instance, a real car may be modeled by a class in a model called Car and classes in the model are modeled
532 in the metamodel by a class called Class. Among other things, metamodel class for Class specifies that a Class
533 has a name 'Car'.

534 **6.2 Concepts**

535 **6.2.1 Inheritance**

536 The familiar concept of generalization is used to denote incorporation of one metaelement into another.
537 Incorporation of concepts results in a new, independent metaelement that 'inherits' the associations,
538 operations, and attributes of the inherited metaelement.

539 A metaelement may directly inherit from at most one other metaelement. It may not inherit from itself
540 either directly or indirectly.

541 All inherited properties, operations, and associations must have unique names in the context of each
542 inheriting metaelement.

543 **6.2.2 Multiplicity**

544 For metaelements that can have values, multiplicity describes the number of values that a metaelement is
545 allowed to have.

546 The allowed number of values is specified by a lower and upper bound.

547 The lower bound is a non-negative integer, with a default value of one (1). The upper bound is an
548 unlimited natural, which is either a non-negative integer or an asterisk "*" specifying an unlimited upper
549 bound. The default upper bound is one (1).

550 In diagrams, if both lower and upper bounds are listed, they are separated by a "..". If only one is
551 shown, it is assumed that it represents both the upper and lower bounds, except if an asterisk is used
552 alone, a lower bound of zero is assumed, (i.e. 0..*).

553 In diagrams, multiplicity is shown within brackets following an attribute or is shown without brackets if
554 shown on an association end. For examples, see Figure 1.

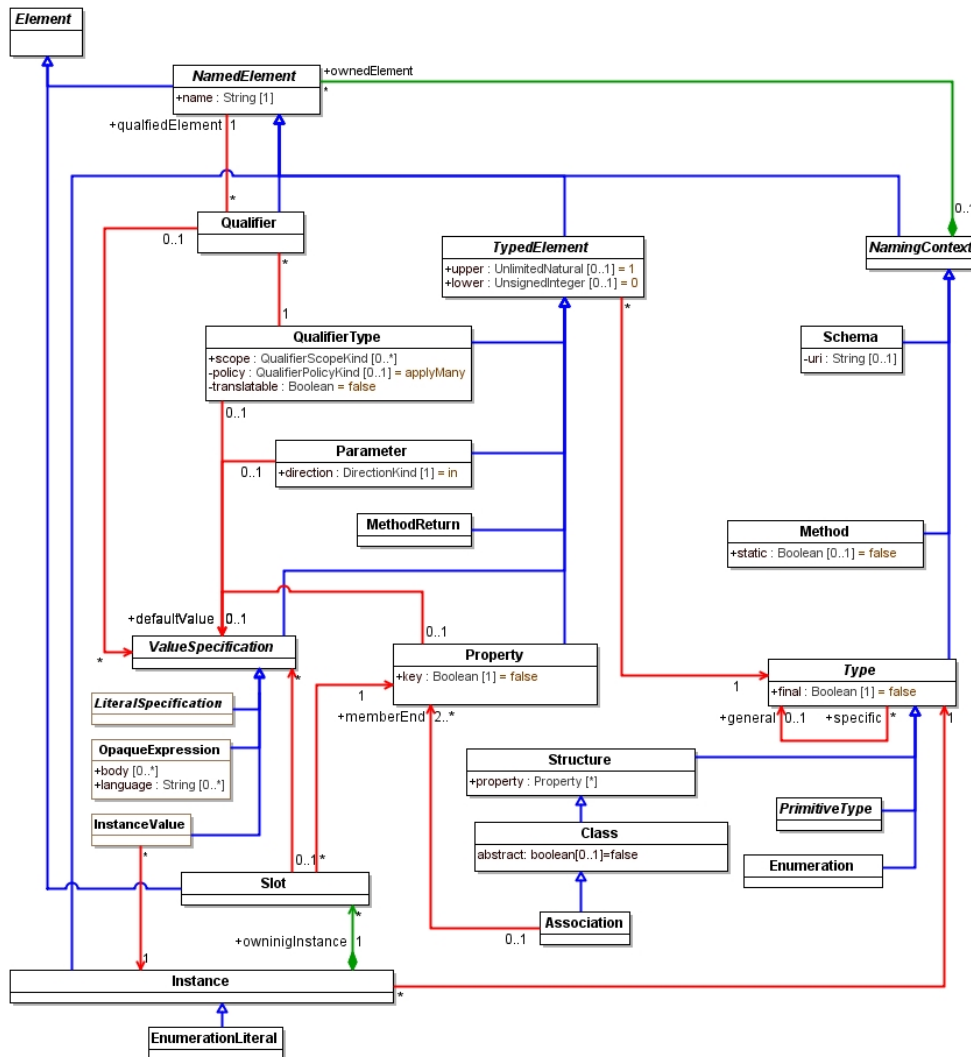
555 **6.2.3 Association**

556 A relationship between metaelements is modeled as an association.

- 557 Each related metaelement is named by an association end. Only binary associations are supported by
558 the metamodel.
- 559 Each association end is represented by an attribute that references the metaclass that it associates. If
560 the association is navigable in the direction of the referenced metaclass that association end is an
561 attribute of the referencing metaclass. Otherwise the attribute representing the association end belongs
562 to the association itself.
- 563 By default, each attribute representing an association end is named by the name of the referenced
564 metaclass with the first letter made lowercase. Like any attribute, the attribute name of each association
565 end must be unique in the metaclass or association to which it belongs.

566 **6.3 Metamodel overview**

567 The principal elements of the CIM Metamodel are shown in Figure 1. See Clause 6.4 for complete details
 568 of each element shown in Figure 1.



569 **Figure 1: Overview CIM Metamodel Diagram**
 570

571 Following is a brief description of each of these elements.

- 572 1) Element is the ancestor of all other metamodels
- 573 2) A Slot specifies the values of a Property in an Instance.
- 574 3) A NamedElement is the ancestor of all metamodels that can be identified by a name

- 575 4) A NamingContext is the ancestor of all metaelements that define the naming rules for other
576 NamedElements.
- 577 5) An Instance specifies the realization of an element.
- 578 6) A Qualifier specifies an extension to a metaelement represented by a named value.
- 579 7) A TypedElement is a named ancestor of metaelements that specify a value or values that conform
580 to a Type
- 581 8) A Method specifies a named behavior in the context of a Class.
- 582 9) A Type specifies an the allowed range of values and the naming rules for component elements.
- 583 10) A Schema contains and provides naming rules for QualifierTypes, other Schemas, Instances and
584 Types
- 585 11) A QualifierType specifies a set of Qualifiers that each provide an extension to a metaelement.
- 586 12) A Parameter is a named value (or collectin of values) passed into or out of a Method
- 587 13) A Property is a named value (orcollection of values) that is a component of a Structure.
- 588 14) A ValueSpecification is a named specification of a value.
- 589 15) An Enumeration is a Type defined by a set of EnumerationLiterals.
- 590 16) A PrimitiveType is a predefined Type.
- 591 17) A Structure is a Type that has zero or more component Properties
- 592 18) A Class is a Structure that has zero or more owned Methods and may additionally have component
593 Structures and Enumerations.
- 594 19) An Association is a Class that represents a relationship between two or more Class Instances
595 specified by two or more Properties of Type Reference.

596 **6.4 Principal metaelements**

597 **6.4.1 CIMM::Association**

598 An Association is both a Class and a relationship between two or more referenced Class instances. As a
599 Class, it defines a set of Properties and Methods that belong to the relationship itself and not to the
600 referenced classes. In particular, two or more of those Properties with a type of Reference are
601 designated as memberEnds in the metamodel. The value of each memberEnd Property is a reference to
602 a related Instance.

603 The specialization and refinement rules defined for Class are applicable to Association.

604 For an Instance of an Association, each memberEnd Property contains a reference to at most one
605 Instance.

606 Each Association Instance must have at least two non-NULL memberEnds. Additionally, each
607 memberEnd with a lower > 0 must be non-NULL.

608 The multiplicity of an association end constrains the size of a collection of Instances referenced by that
609 end when all other ends refer to specific Instances.

610 Note: If an end of an n-ary Association has a lower multiplicity of 1 (or more) then at least that many Instances of
611 that Association must exist with distinct values for memberEnd representing that end.

612 Navigation between an Instance referenced by one MemberEnd to another instance referenced by
613 another MemberEnd of the same Association Instance is possible, but does not specify the efficiency of
614 such navigation.

615 **Generalizations**

- 616 • CIMM::Class (see 6.4.2)

617 **Attributes**

- 618 • No additional attributes

619 **Associations**

- 620 • Each end represents participation of Instances of the Class in relationships defined by the Association.

622 *memberEnd : Property [2..*]*

623 **Constraints**

- 624 • An Association shall not specialize a concrete Class

625 *general->NotEmpty() and not general.oclIsKindOf(Association)*
626 *implies not general.abstract*

- 627 • An Association specializing another concrete Association shall have the same number of ends

628 *general->notEmpty()and*
629 *general.oclIsKindOf(Association) and*
630 *not general->abstract*
631 *implies general.memberEnd->size() = memberEnd->size()*

- 632 • An Association cannot be defined between itself and something else.

633 *self.memberEnd.type.class->excludes(self) and*
634 *self.memberEnd.type.class ->collect(et|et.allparents()->excludes(self))*

- 635 • An Association shall have two or more memberEnds

636 *memberEnd.size() >= 2*

- 637 • An Association shall own all of its memberEnds

638 *property->includesAll(memberEnd)*

639 **6.4.2 CIMM::Class**

640 A Class specifies a classification of Instances and specifies the properties and methods that characterize
641 the state and behavior of those Instances.

642 A Class represents an aspect of a modeled element. Abstract classes represent an incomplete definition
643 of that aspect. Instances of non-abstract classes represent an aspect of an Instance of the modeled
644 element.

645 A Class may have Methods that represent exposed behaviors, and Properties that represent exposed
646 state or status of the modeled element.

647 A Class may participate in Associations as the type of a Property that is an associationEnd of an
648 Association.

649 A Class may include the definition of zero or more Enumerations or Structures.

650 **Generalizations**

- 651 • CIMM::Type (see 6.4.22)

652 **Attributes**

- 653 • No additional attributes

654 **Associations**655 **Constraints**

- 656 • Class is the NamingContext for Method, Property, Enumeration, and Structure

657 *ownedMember.notEmpty implies*

658 *ownedMember.oclsKindOf(Method or Property or Enumeration or Structure)*

- 659 • If not abstract, then one property shall be designated as a Key

660 *If abstract then true*

661 *else*

662 *count(exposedProperty()->key) = 1*

663 *endif*

- 664 • An association may not be the general type of a class.

665 *general.notEmpty() implies not general.oclsKindOf(Association)*

- 666 • The value of the name attribute (i.e., the class name) shall follow the formal syntax defined here:

667 *nestingType.size()= 0 and not oclsKindOf(PrimitiveType)*

668 *implies name.isValidGlobalElementName()*

669 **6.4.3 CIMM::Element**

670 Element is common to all other metaelements.

671 Element is an abstract metaclass.

672 **Generalizations**

- 673 • None

674 **Attributes**

- 675 • No additional attributes

676 **Associations**

- 677 • No additional associations

678 **Constraints**

- 679 • No additional constraints

680 **6.4.4 CIMM::Enumeration**

681 An Enumeration is a type that has a value range defined by the collection of the values of its

682 EnumerationLiterals. Each EnumerationLiteral is explicitly declared, with a name and value. The value of

683 each EnumerationLiteral must conform to the Enumeration's type.

684 Each value of type Enumeration corresponds to the value of exactly one of its EnumerationLiterals.

685 **Generalizations**

- 686 • CIMM::Type (see 6.4.22)

687 **Attributes**

- 688 • No additional attributes

689 **Associations**

- 690 • No additional associations

691 **Constraints**

- 692 • Enumeration is the NamingContext for EnumerationLiteral
693 *ownedMember.notEmpty implies ownedMember.oclIsKindOf(EnumerationLiteral)*
- 694 • Parent type must be IntegerString, or another Enumeration
695 *general.oclIsKindOf(Integer OR String OR Enumeration)*

696 **6.4.5 CIMM::EnumerationLiteral**

697 Represents a possible value for an Enumeration.

698 The name of an EnumerationLiteral shall be unique within its Enumeration. These names are not global
699 and must be qualified by the Enumeration for general use.

700 **Generalizations**

- 701 • CIMM::Instance (see 6.4.6)

702 **Attributes**

- 703 • No additional attributes

704 **Associations**

- 705 • No additional associations

706 **Constraints**

- 707 • The namingContext shall be an Enumeration
708 *namingContext->oclIsKindOf(Enumeration)*
- 709 • The type of an EnumerationLiteral is consistent with the type of the Enumeration.
710 *type = oclIsKindOf(namingContext)*

711 **6.4.6 CIMM::Instance**

712 An Instance specifies the information necessary to realize a Type (including Structure, PrimitiveType,
713 Enumeration, Class and Association) and specifies the existence of an element of the model that
714 conforms to that Structure.

715 If the type is a Structure, at most one slot of the Instance may be used to represent each Property of that
716 Structure. If not all Properties are represented by a slot, then the specification of the instance is
717 incomplete (or partial). If the specification is incomplete, the modeled system shall supply missing
718 information on realization.

719 With appropriate values for input Parameters, Methods can be invoked on an Instance of a Class.

720 Method invocations may cause changes in value to the Properties of Instances in the scope of the
721 Method's execution.

722 Method invocations may cause the creation and deletion of Instances.

723 It is important to keep in mind that when Instance is realized as a model element, it should not be
 724 confused with the actual instance that it is modeling. Therefore, one should not expect the dynamic
 725 semantics of Instance in a model repository to conform to all of the semantics of the elements that they
 726 model.

727 **Generalizations**

- 728 • CIMM::NamedElement (see □)

729 **Attributes**

- 730 • No additional attributes

731 **Associations**

- 732 • A slot giving the value or values of a Property of the instance. An Instance can have one slot per
 733 Property of its type, including inherited properties. It is not necessary to model a slot for each
 734 Property, in which case the Instance is a partial description.

735 *slot: Slot [0..*]*

- 736 • The realizing Type.

737 *type: Type [1]*

738 **Constraints**

- 739 • If the realizing Type is a Class, it shall not be abstract and the namingContext shall be a Schema

740 *type->oclIsKindOf(Class) implies*

741 *not type.abstract and*

742 *namingContext->oclIsKindOf(Schema)*

743

744 **6.4.7 CIMM::InstanceValue**

745 An instance value specifies the value modeled by an instance specification.

746 **Generalizations**

- 747 • CIMM::ValueSpecification (see 6.4.24)

748 **Attributes**

- 749 • No additional attributes

750 **Associations**

- 751 • The instance that is the specified value

752 *instance: Instance [1]*

753 **Constraints**

- 754 • No additional constraints

755 **6.4.8 CIMM::LiteralSpecification**

756 A LiteralSpecification represents the specification of zero or more values, each in the range of a particular
 757 type.

758 LiteralSpecification is an abstract metaclass.

759 LiteralSpecification the concrete subclasses shown in Table 4.

760 **Table 2: Specializations of LiteralSpecification**

761 **Generalizations**

- 762 • CIMM::ValueSpecification (see 6.4.24)

763 **Attributes**

- 764 • No additional attributes

765 **Associations**

- 766 • No additional associations

767 **Constraints**

- 768 • No additional constraints

769

770 **6.4.9 CIMM::Method**

771 A Method represents the ability to invoke a specified behavior of a Class.

772 A Method may have zero or more Parameters that each specify type and multiplicity of values for input
773 arguments or values of output results of a Method invocation.

774 A Method may have at most one methodReturn that specifies the type and multiplicity of values of an
775 additional output of a Method invocation.

776 Methods of a Class are inherited by each sub Class of that Class in addition to the Methods defined in the
777 sub Class.

778 The set of Methods defined in a Class and together with the exposed Methods inherited from its super
779 Class, (if it has one), contains all of the Methods exposed by the Class.

780 A Class defining a Method may indicate that the Method overrides an inherited Method. In this case, the
781 Class exposes only the overriding Method. The characteristics of the overriding Method are formed by
782 using the characteristics of the overridden Method as a basis and changing them as defined in the
783 overriding Method, within certain limits as defined in the 'Constraints' clause below.

784 If static is true, the Method does not depend on any per-instance data. Non-static Methods are invoked in
785 the context of an Instance; for static Methods, the context of a Class is sufficient.

786 **Generalizations**

- 787 • CIMM::NamingContext (see 6.4.12)

788 **Attributes**

- 789 • If static is true the method may be invoked on the Class declaration in the model. If static is false,
790 then the Method may only be invoked on a modeled Instance of the Class.

791 *static : boolean [0..1] = false*

792 **Associations**

- 793 • A Method that is overridden by this Method

794 *overridden : Method [0..1]*

- 795 • The Class that provides the NamingContext for the overridden Method

796 *overrideContext : Class [0..1]*

797 Constraints

- 798 • The namingContext shall be a Class

799 *namingContext->oclIsKindOf(Class)*

- 800 • Method is the NamingContext for MethodReturn and Parameter

801 *ownedMember.notEmpty implies ownedMember.oclIsKindOf(MethodReturn or Parameter)*

- 802 • Method may have at most one MethodReturn

803 *ownedMember->select(oclIsKindOf(MethodReturn)).size() <= 1*

- 804 • A Method may only redefine a Method of the same name.

805 *overridden->notEmpty() implies*

806 *overridden.oclIsKindOf(Method) and name = overridden.name*

- 807 • An overriding Method shall have the same signature (i.e., Parameters and MethodReturn) as the
808 Method it redefines, with exception that additional out Parameters are allowed and additional in or
809 inout Parameters are allowed if a default value is specified.

810 *overridden.size() = 1 implies*

811 *(methodReturn.size() = 1 implies*

812 *overridden.methodReturn.size() = 1 and*

813 *methodReturn.type->oclIsKindOf(overridden.parameter.type)*

814 *) and*

815 *(parameter.size() >= overridden.parameter.size() and*

816 *Sequence{1 .. overridden.parameter.size()->*

817 *forall (i |*

818 *parameter.name->at(i) = overridden.parameter.name->at(i) and*

819 *parameter.type->at(i)->*

820 *oclIsKindOf(overridden.parameter.type->at(i) and*

821 *parameter.direction->at(i) =*

822 *overridden.parameter.direction->at(i)) and*

823 *Set { overridden.parameter.size() .. parameter.size()->*

824 *forall (i | parameter.direction = not DirectionKind::out implies*

825 *parameter.defaultValue.size() = 1)*

826 *)*

827 6.4.10 CIMM::MethodReturn

828 A MethodReturn specifies the Type an optional return result of a method . The Type and multiplicity of the
829 MethodReturn restricts what values may be returned.

830 Generalizations

- 831 • CIMM::TypedElement (see 6.4.23)

832 Attributes

- 833 • No additional attributes

834 **Associations**

- 835 • The Method that this MethodReturn belongs to.

836 *method: Method [1]*

- 837 • A MethodReturn that is overridden by this MethodReturn

838 *overridden: MethodReturnm [0..1]*

- 839 • The Method that provides the NamingContext for the overridden MethodReturn

840 *overrideContext: Method [0..1]*

841 **Constraints**

- 842 • The NamingContext shall be a Method

843 *namingContext->oclIsKindOf(Method)*

844 **6.4.11 CIMM::NamedElement**

845 A NamedElement has a name.

846 If the associated NamingContext is present, the value of the name attribute identifies the named element
847 within the associated NamingContext. If the NamingContext association is not present, the value of the
848 name attribute is assumed to be unique within the modeled environment.

849 NamedElement is an abstract metaclass.

850 **Generalizations**

- 851 • CIMM::Element (see 6.4)

852 **Attributes**

- 853 • A name for this element that is unique in the associated NamingContext.

854 *name: String [0..1] = NULL*

855 **Associations**

- 856 • The NamingContext in which the name attribute is unique.

857 *namingContext: NamingContext [0..1]*

- 858 • Qualifiers that extend this NamedElement

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

860 **Constraints**

- 861 • No additional constraints

862 **Operations**

- 863 • The query isDistinguishableFrom() determines whether two NamedElements may logically co-exist
864 within a NamingContext. Two named elements are distinguishable if:

- 865 • they have unrelated types or
- 866 • they have related types but different names.

```
867 NamedElement::isDistinguishableFrom(n:NamedElement, ns: NamingContext): Boolean; isDistinguishable =  
868 if self.oclIsKindOf(n.oclType) or n.oclIsKindOf(self.oclType)  
869 then self.name <> n.name  
870 else true
```

- 871 *endif*
- 872 • Tests whether the string is a valid CIM element name and returns true if that is the case, else false.
873 A valid CIM element names as follows:
- 874 • The name of a CIM element shall consist of the following characters: "_", "A", ..., "Z", "a", ...,
875 "z", "0", ..., "9", U+0080, ..., U+FFEF
- 876 • The name of a CIM element shall start with one of the following characters: "_", "A", ..., "Z", "a",
877 ..., "z", U+0080, ..., U+FFEF
- 878 • The name of a CIM element shall be at least one character long.
- 879 • These rules apply to the names of all CIM element types.
- 880 • Some CIM element types have additional rules. For example, the name of a CIM class shall
881 have an underscore after its schema name part. Such additional rules are validated by
882 additional OCL constraints.

```

883 String::isValidElementName() : Boolean
884 let initial : String = Sequence('_', 'A'..'Z', 'a'..'z', \u0080..\uFFEF) in
885 let subsequent : String =
886     Sequence('_', 'A'..'Z', 'a'..'z', '0'..'9', \u0080..\uFFEF) in
887 let namesize : Integer = self.size() -- The size of the className in
888     -- position of first '_' in class name
889 if namesize = 0 then false
890 else if namesize() = 1
891     then -- name must be one of the initial characters
892         initial->count(self)=1
893     else -- first character of name must be one of the initial characters
894         if initial->count(self.at(1)) <> 1 then false
895         else -- the rest of the characters must be a subsequent character
896             Sequence { 2 .. namesize }->
897                 forAll( i | subsequent->count(self.at(i)) = 1)
898         endif
899     endif
900 endif
    
```

- 901 • A valid global element name has the abnf schemaName "_" elementName, where schemaName and
902 elementName are both valid element names. And schemaName does not contain an "_".

```

903 String::isValidGlobalElementName() : Boolean
904 let cnsz : Integer = self.size() -- The size of the name
905 in
906     |-- position of first '_' in name
907 let upos : Integer = Sequence { 1 .. cnsz }->
908     any( i | self.substring( i, i) = '_' )
909 in
910     -- If '_' is not found, that's an error
911 if upos.oclIsUndefined()
912 then false |
913 else
914     let schema : String = /* schema name */
915         self.substring( 1, upos - 1)
916     in
917     let sname : String = /* local name */
    
```

Comment [GME1]: I'm still of the opinion that we can remove this in favor of name.size();

Need to resolve rules w.r.t. schema name vs top-level Schema (folder) name.

I still think we can separate these two concepts.

Qualified CIM name goes from
CIM::p1::p2::...::CIM_structureName
to
CIM::p1::p2::...::structureName

Then we can enforce uniqueness w.r.t. top-level Schema so that CIM::structureName is required to be unique.

Comment [GME2]: I'm still of the opinion that we can remove this in favor of name.size();

Need to resolve rules w.r.t. schema name vs top-level Schema (folder) name.

I still think we can separate these two concepts.

Qualified CIM name goes from
CIM::p1::p2::...::CIM_structureName
to
CIM::p1::p2::...::structureName

Then we can enforce uniqueness w.r.t. top-level Schema so that CIM::structureName is required to be unique.

```

918         self.substring( upos + 1, cnsiz)
919     in
920         schema.isValidElementName() and
921         sname.isValidElementName()
922     endif

```

923 6.4.12 CIMM::NamingContext

924 An element that is a NamingContext provides a context in which the names of member NamedElements
925 shall be unique.

926 A NamingContext provides a scope in which the names of named elements shall be unique.

927 Two elements are distinguishable if they have unrelated types, or related types but different names.

928 NamingContext is an abstract metaclass.

929 Generalizations

- 930 • CIMM::NamedElement (see 6.4.4)

931 Attributes

- 932 • No additional attributes

933 Associations

- 934 • A collection of NamedElements owned by the NamingContext

935 *ownedMember: NamedElement [*]*

936 Constraints

- 937 • All the members of a NamingContext are distinguishable within it.

938 *ownedMember->forAll(memb | ownedMember->excluding(memb)->*
939 *forAll(other | memb.isDistinguishableFrom(other, self)))*

940 6.4.13 CIMM::OpaqueExpression

941 An OpaqueExpression contains language-specific text strings used to describe a value or values, and an
942 optional specification of the languages. One predefined language for specifying expressions is OCL.
943 Natural language or programming languages may also be used.

944 Generalizations

- 945 • CIMM::ValueSpecification (see 6.4.24)

946 Attributes

- 947 • The text of the expression, possibly in multiple languages.

948 *body: String [0..*]*

- 949 • Specifies the languages in which the expression is stated. The interpretation of the expression body
950 depends on the language. If the languages are unspecified, they might be implicit from the
951 expression body or the context. Languages are matched to body strings by order.

952 *language: String [0..*]*

953 **Associations**

- 954 • No additional associations

955 **Constraints**

- 956 • If the language attribute is not empty, then the size of the body and language arrays must be the
957 same.

958 *language->notEmpty() implies (body->size() = language->size())*

959 **6.4.14 CIMM::Parameter**

960 A Parameter specifies the type, multiplicity, and values associated with an input argument to a Method
961 invocation or an output argument of a method. Note that an optional return result is specified via
962 MethodResult.

963 The direction attribute specifies whether the Parameters value is passed into, out of, or both into and out
964 of the Method.

965 If a default is specified for an input Parameter and if and only if no value or values are supplied at
966 invocation of the Method then the default value shall be evaluated and used to specify the value or values
967 of the Parameter.

968 If a default is specified for an output Parameter and if and only if no argument is supplied at termination of
969 the Method then the default value shall be evaluated and used to specify the value or values of the
970 Parameter.

971 **Generalizations**

- 972 • CIMM::TypedElement (see 6.4.23)

973 **Attributes**

- 974 • Indicates whether a Parameter is being sent into or out of a Method. Possible values are in, inout,
975 and out.

976 *direction : DirectionKind [1] = DirectionKind::in*

977 **Associations**

- 978 • An optional specification of the default value or values.

979 *defaultValue: ValueSpecification [0..1]*

- 980 • The method that this parameter belongs to.

981 *method: Method [1]*

- 982 • A Parameter that is overridden by this Parameter

983 *overridden : Parameter [0..1]*

- 984 • The Method that provides the NamingContext for the overridden Parameter

985 *overrideContext : Method [0..1]*

986 **Constraints**

- 987 • The NamingContext shall be a Method

988 *namingContext->oclIsKindOf(Method)*

- 989 • Parameter is a NamingContext for ValueSpecification

990 *ownedMember.notEmpty implies ownedMember.oclIsKindOf(ValueSpecification)*

991 **6.4.15 CIMM::PrimitiveType**

992 A predefined Type. Instances of a PrimitiveType are data values. The values are in many-to-one
 993 correspondence to mathematical elements defined outside of the CIM metamodel (for example, the
 994 various integers). Instances of primitive types do not have identity. If two instances have the same
 995 representation, then they are indistinguishable.

996 The PrimitiveTypes of the metamodel are specified in Table 3.

997

Table 3: Predefined Types

Predefined Type	Generalizes	Abstract	Interpretation
AggregationKind	PrimitiveType	false	<p>Enumeration for specifying the aggregation of a referenced element into a referencing element.</p> <ul style="list-style-type: none"> no aggregation. <i>none</i> shared aggregation. <i>shared</i> composite aggregation, i.e., the composite element has responsibility for the existence and storage of the composed elements. <i>composite</i>
ArrayKind	PrimitiveType	false	<p>Enumeration for specifying the characteristics of the set of values belonging to an array.</p> <ul style="list-style-type: none"> Indicates the set of values is a bag that may contain duplicates and order is not preserved. (Equivalent to OCL::BagType.) <i>bag</i> Indicates the set of values is a bag that may not contain duplicates and order is not preserved. (Equivalent to OCL::SetType.) <i>set</i> Indicates the set of values is ordered and may not contain duplicates, and that order is not preserved across adds, deletes, or updates, (including creation.) (Equivalent to OCL::OrderedSetType.) <i>orderedSet</i> Indicates the set of values is ordered and may contain duplicates, and that the index of each value is preserved, even if the value of the indexed entry is changed. (Equivalent to OCL::SequenceType.) <i>indexed</i>
Boolean	PrimitiveType	false	Boolean
Datetime	String	false	A string containing a date-time

Predefined Type	Generalizes	Abstract	Interpretation
DirectionKind	PrimitiveType	false	<p>An enumeration used to specify direction of parameters.</p> <ul style="list-style-type: none"> Indicates that Parameter values are passed into the Method by the caller. <i>in</i> Indicates that Parameter values are passed into a Method by the caller and then back out to the caller. <i>inout</i> Indicates that Parameter values are passed from a Method out to the caller. <i>out</i>
Duration	DateTime	true	<p>Duration values have a range from zero to one microsecond less than 100,000,000 days, with a granularity of microseconds. The last second of that range is reserved, with the value of one second less than 100,000,000 days reserved to indicate an unlimited duration.</p>
Integer	Numeric	true	<p>An element in the (infinite) set of integers (...-2, -1, 0, 1, 2...). It is used as a type for integer properties and expressions.</p>
Numeric	PrimitiveType	true	<p>A Numeric element is a real or integer number in the continuum (...-2, -1, 0, 1, 2...).</p>
OctetString	String	false	<p>A string of binary data of a specified length.</p>
QualifierPolicyKind	PrimitiveType	false	<p>Enumeration for defining Qualifier value change policies.</p> <ul style="list-style-type: none"> Indicates a Qualifier's value may be explicitly set many times and that value is propagated to corresponding elements in sub types. If not explicitly set, (directly or indirectly), a Qualifier's value is the default value specified by the QualifierType that defines the Qualifier. <i>applyMany</i> Indicates a Qualifier's value may be explicitly set once after initialization and that value is propagated to corresponding elements in sub types. If not explicitly, (directly or indirectly), a Qualifier's value is the default value specified by the QualifierType that defines the Qualifier. <i>applyOnce</i> may be explicitly set many times but that value is not propagated to corresponding elements in sub types. If not explicitly set, (directly or indirectly), a Qualifier's value is the default value specified by the QualifierType that defines the Qualifier. <i>restricted</i>

Predefined Type	Generalizes	Abstract	Interpretation
QualifierScopeKind	PrimitiveType	false	<p>Enumeration for specifying the metaelements that may be extended by a QualifierType.</p> <ul style="list-style-type: none"> Each literal specifies a metaelement that may be extended. <p><i>Association</i> <i>Class</i> <i>Enumeration</i> <i>EnumerationLiteral</i> <i>Method</i> <i>Parameter</i> <i>Property</i> <i>QualifierType</i> <i>Structure</i></p>
Real	Numeric	true	A real represents a numeric value along a continuum, such as -7, 1/3, or 9.4 where the precision of the representation is limited by the size of the storage dedicated to the type.
Real32	Numeric	false	4-byte floating-point compatible with IEEE-754® Single format
Real64	Numeric	false	8-byte floating-point compatible with IEEE-754® Double format
Real128	Numeric	false	16-byte floating-point compatible with IEEE-754® Quadruple format
Reference	PrimitiveType	false	The address of an instance of a class.
SignedInteger	Integer	false	An Integer in the set of integers (...-2, -1, 0, 1, 2...).
Sint8	SignedInteger	false	Signed 8-bit integer
Sint16	SignedInteger	false	Signed 16-bit integer
Sint32	SignedInteger	false	Signed 32-bit integer
Sint64	SignedInteger	false	Signed 64-bit integer
Sint128	SignedInteger	false	Signed 128-bit integer
String	PrimitiveType	false	A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters. An instance of String defines a sequence of text. The semantics depends on its use, it can be a comment, computational language expression, OCL expression, etc. It is used as a type for String properties and expressions. Non-NULL string typed values shall contain zero or more UCS characters (see Annex D).

Predefined Type	Generalizes	Abstract	Interpretation
TimeStamp	DateTime	false	<p>Timestamps represent a Coordinated Universal Time (UTC), see Annex B.</p> <p>A Timestamp includes a time zone correction field that specifies an integral number of hours before or after Greenwich Mean Time (GMT), also known Zulu time. A positive time zone correction factor is used for time zones east of Greenwich, and a negative time zone correction factor is used for time zones west of Greenwich. The range of the time zone correction factor is between -999..+999</p> <p>The resolution of the timestamp is years (0..9999), month in year (1..12), days in month (1..31), hours in day (0..23), minutes in hour (0..60), seconds in minute (0..60), and microseconds in second (0..999,999).</p> <p>A value of 00000101000000.000000 at datetime offset +720 is equivalent to January 1st, of 1 BCE at 00:00:00.000000 hours at a time zone offset of 720 hours to the east of Greenwich. This definition implicitly performs timestamp normalization.</p> <p>NOTE: 1 BCE is the year before 1 CE based on the proleptic Gregorian calendar.</p>
UInt8	UnsignedInteger	false	Unsigned 8-bit integer
UInt16	UnsignedInteger	false	Unsigned 16-bit integer
UInt32	UnsignedInteger	false	Unsigned 32-bit integer
UInt64	UnsignedInteger	false	Unsigned 64-bit integer
UInt128	UnsignedInteger	false	Unsigned 128-bit integer
UnlimitedNatural	UnsignedInteger	false	An element in the set of non-negative integers (0, 1, 2...) and unlimited. The value of unlimited is shown using an asterisk (*).
UnsignedInteger	Integer	false	An Integer in the set of non-negative integers (0, 1, 2...).

998 **Generalizes**

- 999 • CIMM::Type (see 6.4.22)

1000 **Attributes**

- 1001 • No additional attributes.

1002 **Associations**

- 1003 • No additional associations.

1004 **Constraints**

- 1005 • No additional constraints.

1006 **6.4.16 CIMM::Property**

1007 A Property is represents an attribute of a Structure. It relates an Instance to a value or collection of
1008 values of the Type of the Property.

1009 Properties of a type (i.e an association, a class or a structure), are inherited by more specific types (i.e.
1010 sub structures or sub classes). Instances of the specialized type have the inherited properties in addition
1011 to the properties defined in the specialized type. The combined set of properties are referred to as the
1012 exposed properties of the type.

1013 A sub type shall not define a property with the same name as an inherited property without overriding that
1014 property. The type containing the overridden property shall be a super type of type containing the
1015 overriding property. .

1016 When a property overrides an inherited property, the type containing the overriding property exposes only
1017 the overriding property. The characteristics of the overriding property are formed by using the
1018 characteristics of the overridden property as a basis, changing them as defined in the overriding property,
1019 within certain limits as defined in constraints section below.

1020 When an Instance containing a property is instantiated, if no other value is supplied by any means and if
1021 a property defines a default value, that default value is evaluated to provide a value for the Property,
1022 otherwise no value is assigned to the property, (i.e. the property value will be NULL.)

1023 **Generalizations**

- 1024 • CIMM::TypedElement (see 6.4.23)

1025 **Attributes**

- 1026 • If true, the value of this Property can be used to uniquely identify an instance of the containing Class.
1027 *key : Boolean = false*

1028 **Associations**

- 1029 • A specification of a string that is evaluated to give a default value for the Property when instantiated.
1030 *defaultValue : ValueSpecification [0..1]*
- 1031 • A Property that overrides this Property
1032 *overridingElement : Property [*]*
- 1033 • A Property that is overridden by this Property
1034 *overridden : Property [0..1]*
- 1035 • The Structure that provides the NamingContext for the overridden Property
1036 *overrideContext : Structure [0..1]*
- 1037 • Slot holds values of Property in an Instance.
1038 *slot : Slot [0..*]*
- 1039 • The type that owns the Property.
1040 *structure : Structure [1]*

1041 **Constraints**

- 1042 • The NamingContext shall be a Structure
1043 *namingContext->oclIsKindOf(Structure)*
- 1044 • Property is a NamingContext for ValueSpecification
1045 *ownedMember.notEmpty implies ownedMember.oclIsKindOf(ValueSpecification)*

- 1046 • An overridden property must be inherited from a more general type containing the redefining
1047 property.

```
1048   if overridden->notEmpty() then
1049     (overrideContext->notEmpty() and
1050      overridden->
1051       forAll( rp | ((overrideContext->
1052        collect(fc| fc.allParents()))->asSet())->
1053         collect(c| c.allProperties())->asSet())-> includes(rp))
```

- 1054 • An overriding Property shall have the same name as the Property it redefines.

```
1055 overridden->notEmpty() implies name = overridden.name
```

- 1056 • An overriding property shall have type that is consistent with the Property it redefines.

```
1057 overridden->notEmpty() implies isConsistentWith(overridden)
```

- 1058 • A key property may not be modified and must be of primitiveType and is not null.

```
1059 key = true implies (readOnly = true) and
1060 self.oclIsKindOf(PrimitiveType) and
1061 lower = 1 and upper = 1
```

1062 Operations

- 1063 • The query isConsistentWith() specifies, for any two Properties in a context in which redefinition is
1064 possible, whether redefinition would be logically consistent. A redefining property is consistent with a
1065 redefined property if the type of the redefining property conforms to the type of the redefined
1066 property, and the multiplicity of the redefining property (if specified) is contained in the multiplicity of
1067 the redefined property.

```
1068 Property::isConsistentWith(redefinee : Property) : Boolean pre:  redefinee.isOverrideContextValid(self)
1069 isConsistentWith = redefinee.oclIsKindOf(Property) and
1070 let prop : Property = redefinee.oclAsType(Property)
1071 in
1072   (prop.type.conformsTo(self.type) and
1073    ((prop.lowerBound()->notEmpty() and self.lowerBound()->notEmpty())
1074     implies
1075      prop.lowerBound() >= self.lowerBound()) and
1076    ((prop.upperBound()->notEmpty() and self.upperBound()->notEmpty())
1077     implies
1078      prop.lowerBound() <= self.lowerBound()))
```

1079 6.4.17 CIMM::Qualifier

1080 A Qualifier represents an extension attribute of a metaelement. Qualifiers are defined by a QualifierType.

- 1081 • The name of any localized qualifiers shall conform to the following formal syntax defined in ABNF:

```
1082 localized-qualifier-name = qualifier-name '_' locale
1083 locale = language-code '_' country code
1084 ; the locale of the localized qualifier
```

- 1085 • Where:

- 1086 • qualifier-name is the name of the associated QualifierType

- 1087 • language-code is a language code as defined in ISO 639-1:2002, ISO 639-2:1996, or ISO 639-
1088 3:2007

- 1089 • country-code is a country code as defined in ISO 3166-1:2006, ISO 3166-2:2007, or ISO 3166-
1090 3:1999
- 1091 EXAMPLE: For the base qualifier named Description, the localized qualifier for Mexican Spanish language is
1092 named Description_es_MX.
- 1093 • The string value of a localized qualifier shall be a translation of the string value of its base qualifier
1094 from the language identified by the locale of the base qualifier into the language identified by the
1095 locale specified in the name of the localized qualifier.
- 1096 •
- 1097 The effective value of a Qualifier is its value in the current element. This depends on the value of the
1098 QualifierType.policy attribute.
- 1099 The effective value of a Qualifier starts with the default value defined by QualifierType.
- 1100 If a value is then assigned to that Qualifier, or on a decendent of that element, then the effective value for
1101 the Qualifier on that element is as specified.
- 1102 The value that is then propagated to decendents of that element depends on the Flavor. If Restricted,
1103 then the defaultValue of the Qualifier is propagated to decendents. Otherwise, the new value is
1104 propagated.
- 1105 The ancestry of an element is the set of elements that results from recursively determining its ancestor
1106 elements. An element is not considered part of its ancestry.
- 1107 The ancestor of an element depends on the kind of element, as follows:
- 1108 • For a class, its superclass is its ancestor element. If the class does not have a superclass, it has no
1109 ancestor.
- 1110 • For a property (including references) or method, the overridden element is its ancestor. If the
1111 element is not overriding another element, it does not have an ancestor.
- 1112 • For a parameter of a method, the like-named parameter of the overridden method is its ancestor. If
1113 the method is not overriding another method, its parameters do not have an ancestor.
- 1114 •
- 1115 **Generalizations**
- 1116 • NamedElement (see 6.4.4)
- 1117 **Attributes**
- 1118 • No additional attributes
- 1119 **Associations**
- 1120 • The elements that this Qualifier is scoped to.
1121 *qualifiedElement : NamedElement [1..*]*
- 1122 • The defining QualifierType.
1123 *qualifierType : QualifierType [1]*
- 1124 • The values of the Qualifier.
1125 *value : Value [0..*]*
- 1126 **Constraints**
- 1127 • The name of a Qualifier shall match the name of the QualifierType

```

1128 ((not qualifierType.translatable) and qualifierType.name = name)
1129 Or -- the localized prefix must match...
1130 (qualifierType.translatable and
1131 qualifierType.name = name->substring(1, qualifierType.name.size()) and
1132 name->substring(qualifierType.name.size()+1,
1133 qualifierType.name.size()+1) = '_' )

```

- If a qualified element redefines another element, then a Qualifier specification must be conformant with QualifierPolicyKind.

```

1136 qualifierType.policy=QualifierPolicyKind::applyOnce implies
1137 qualifierType.Qualifier->select(ql |
1138 ql.value.size() <> qualifierType.defaultValue.size() or
1139 set (1.. ql.values.size()->
1140 select(i | ql.value->at(i) <> qualifierType.defaultValue->at(i))).
1141 size())<=1

```

1142 6.4.18 CIMM::QualifierType

1143 A QualifierType defines an extension attribute of one or more metaelements. Each extension attribute is
1144 modeled as a Qualifier.

1145 Each Qualifier defines additional information, semantics, or behaviors for the qualified metaelement.

1146 The extended metaelements are specified via the QualifierType.scope attribute.

1147 Base model elements are model elements that are not inherited from another model element realized
1148 from the same metaelement.

1149 Each base model element is realized from its corresponding metaelement extended by the set of
1150 Qualifiers defined by all QualifierTypes that are scoped to that metaelement or to one its specializations.
1151 The default value of each Qualifier is defined by the default value specified by the QualifierType. If no
1152 default value is specified, then no value is assigned. (i.e. the Qualifier is NULL.)

1153 The enumerated values of the QualifierType.policy determine how the value of the Qualifier may be
1154 changed, (see 6.4.15) for a description of the QualifierPolicyKind enumerations.

1155 QualifierTypes that have translatable set to true create Qualifiers that support localization. Localized
1156 Qualifiers allow the specification of Qualifier values in a specific language.

1157 Generalizations

- NamedElement (see 6.4.4)

1159 Attributes

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

```
1161 policy : QualifierPolicyKind [1] = CIMM::QualifierPolicyKind::applyMany
```

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

```
1163 scope : QualifierScopeKind [0..*] = NULL
```

- Specifies whether the resultant Qualifiers support localized values

```
1165 translatable : Boolean = false
```

1166 Associations

- Qualifiers defined on this QualifierType

1168 `qualifier : Qualifier [0..*]`

- The default values for this Qualifier

1170 `defaultValue : Value [0..*]`

1171 Constraints

- The NamingContext shall be a Schema

1173 `namingContext->oclIsKindOf(Schema)`

- The name of a QualifierType shall not contain an underscore

1175 `let initial : String = Sequence('A'..'Z', 'a'..'z', '\u0080..\uFFEF') in`

1176 `let subsequent : String =`

1177 `Sequence('A'..'Z', 'a'..'z', '0'..'9', '\u0080..\uFFEF') in`

1178 `let namesize : Integer = name.size() -- The size of the name`

1179 `if namesize = 0 then false`

1180 `else if namesize = 1`

1181 `then -- name must be one of the initial characters`

1182 `initial->count(name)=1`

1183 `else -- first character of name must be one of the initial characters`

1184 `if initial->count(name.at(1)) <> 1 then false`

1185 `else -- the rest of the characters must be a subsequent character`

1186 `Sequence { 2 .. namesize }->`

1187 `forAll(i | subsequent->count(name.at(i)) = 1)`

1188 `endif`

1189 `endif`

1190 `endif`

1191 6.4.19 CIMM::Schema

1192 A Schema provides a NamingContext for Instance, sub Schemas, Types and Values.

1193 Generalizations

- CIMM::NamingContext (see 6.4.12)

1195 Attributes

- URI of the schema definition

1197 `uri: String [0..1]`

1198 Associations

- No additional associations

1200 Constraints

- The value of the name attribute (i.e., the Schema name) shall follow the formal syntax defined here:

1202 `name.isValidElementName()`

- Schema is the NamingContext for Instance, QualifierType, Schema, Type, and ValueSpecification

1204 `ownedMember.notEmpty implies`

1205 `ownedMember.oclIsKindOf(Instance or QualifierType or Schema or Type)`

1206 **Operations**

- 1207 • Determine the fully qualified Schema path of this Schema

```
Context Schema::SchemaPath() String
```

```
post:
```

```
if owningSchema.isEmpty() then result=name
```

```
else result = SchemaPath.concat("::").concat(name)
```

```
endif
```

1213 **6.4.20 CIMM::Slot**

1214 A Slot is utilized by an Instance to specify the values of a Property of the Structure of the Instance.

1215 Each Slot shall contain values, in accordance with the characteristics of the Property, for example its type
1216 and multiplicity, (a lower bound of 0 specifies that the Property may have no values).

1217 For every Property that has a specified default, if an initial value of the Property is not specified, (explicitly
1218 or implicitly), then the default value specification of the Property is evaluated to set the initial value of the
1219 Slot.

1220 **Generalizations**

- 1221 • Element (see 6.4.3)

1222 **Attributes**

- 1223 • No additional attributes

1224 **Associations**

- 1225 • The defining property for the values referenced by the slot.

```
1226 property : Property [1]
```

- 1227 • The Instance that owns this slot.

```
1228 owningInstance : Instance [1]
```

- 1229 • The value or values corresponding to the defining Property.

```
1230 valueSpecification : ValueSpecification [0..*]
```

1231 **Constraints**

- 1232 • No additional constraints

1233 **6.4.21 CIMM::Structure**

1234 A Structure is a Type whose Instances are identified only by their value. A Structure may contain
1235 Properties to support the modeling of structured data types.

1236 An Instance of a Structure must contain values for each property that is a member of that Structure, in
1237 accordance with the characteristics of the Property, for example its Type and multiplicity, (lower bound of
1238 0 implies that the Property may have no values).

1239 When a Structure is instantiated: for each Property of that Structure, if an initial value of the Property is
1240 not supplied for the instantiation; then if defined, the default value specification is evaluated to set the
1241 initial value of the Property for the Instance. If no default value is specified, then no value is assigned to
1242 the Property.

1243 Within a specialization of a Structure, the definition of a Property may be overridden to refer to a more
 1244 specialized Type or to add additional constraints on its values.

1245 **Generalizations**

- 1246 • CIMM::Type (see 6.4.22)

1247 **Attributes**

- 1248 • No additional attributes

1249 **Associations**

- 1250 • No additional associations

1251 **Constraints**

- 1252 • Structure is the NamingContext for Property

1253 *ownedMember.notEmpty implies ownedMember.oclIsKindOf(Property)*

1254 **Operation**

- 1255 • The query allProperties() gives all of the Properties in the namespace of the Structure. In general,
 1256 through inheritance, this will be a larger set than property.

1257 *Structure::allProperties(): Set(Property);*
 1258 *allProperties = property->select(oclIsKindOf(Property))*

- 1259 • The exposedProperty operation excludes overridden properties.

1260 *Structure::exposedProperty(inhs: Set(Property)) : Set(Property);*
 1261 *exposedProperty = inhs->excluding(inh | property->*
 1262 *select(oclIsKindOf(Property))->*
 1263 *select(overridden->includes(inh)))*

1264 **6.4.22 CIMM::Type**

1265 A Type defines an allowable range of values.

1266 A Type is a NamingContext for its members. The specification of the characteristics used to distinguish
 1267 members is deferred to elements that specialize from type.

1268 A Type that is abstract serves only as a base for new Types. It is not possible to create instances of
 1269 abstract types.

1270 Types may be arranged in a generalization hierarchy that represents subtype/supertype relationships
 1271 between Types. The generalization hierarchy is a rooted, directed graph and does not support multiple
 1272 inheritance.

1273 Type is an abstract metaelement.

1274 **Generalizations**

- 1275 • CIMM::NamingContext (see 6.4.12)

1276 **Attributes**

- 1277 • Only Types that are not abstract may be part of an instance.

1278 *abstract : Boolean [1] = false*

- 1279 • Only Types that are not final may be specialized.

1280 `final : Boolean [1] = false`

1281 **Associations**

- 1282 • Specifies a general type. Only single inheritance.

1283 `general : Type [0..1]`

- 1284 • Specifies the specializations of this type

1285 `specific : Type [0..*]`

1286 **Constraints**

- 1287 • The NamingContext shall be a Schema

1288 `namingContext->oclIsKindOf(Schema)`

- 1289 • All generalizations of an abstract type shall be abstract.

1290 `inv: abstract and general->notEmpty() implies general.abstract`

- 1291 • Final types may not be abstract and may not be subclassed.

1292 `final=true implies abstract=false and specific.size()=0`

1293 **Operations**

- 1294 • The query allParents() gives all of the direct and indirect ancestors of a generalized type.
1295 Note: single inheritance for Types.

1296 `Type::allParents(): Set(Type); -- recursively collect parents`

1297 `allParents = self.general->union(self.general->collect(p | p.allparents())`

1298 **6.4.23 CIMM::TypedElement**

1299 A TypedElement has values that conform to a Type. The allowable number of values is known as the
1300 element's multiplicity, (see 6.2.2). If multiplicity has an upper bound of 0 or one, then the element is a
1301 scalar. Otherwise, an instance of the element contains a set of values. The characteristics of the set are
1302 specified by the PrimitiveType enumeration MultiplicityKind (see 6.4.15).

1303 TypedElement is an abstract metaelement.

1304 **Generalizations**

- 1305 • NamedElement (see 6.4.4)

1306 **Attributes**

- 1307 • For a multivalued multiplicity, specifies how values are stored.

1308 `multiplicityKind : CIMM::MultiplicityKind [1] = MultiplicityKind::bag`

- 1309 • Specifies the lower bound of the multiplicity interval, expressed as an unsigned integer.

1310 `lower : UnsignedInteger [1] = 0`

- 1311 • Specifies the upper bound of the multiplicity interval, expressed as an unlimited natural.

1312 `upper : UnlimitedNatural [1] = 1`

1313 **Associations**

- 1314 • Has a Type.

1315 `type: Type [1]`

1316 **Constraints**

- 1317 • The upper bound of a multiplicity shall be greater than or equal to the lower bound.

1318 *upper >= lower*

- 1319 • The upper bound of a scalar shall be one.

1320 *MultiplicityKind = MultiplicityKind::scalar implies upper = 1*

1321 **6.4.24 CIMM::ValueSpecification**

1322 ValueSpecification is used to identify a value or values in a model. It may reference an instance or it may
1323 be an expression denoting an instance or instances when evaluated.

1324 ValueSpecification is an abstract metaelement.

1325 ValueSpecification the concrete subclasses shown in Table 4.

1326 **Table 4: Specializations of ValueSpecification**

Specialization	Interpretation
LiteralBoolean	A Boolean value represented by 'true' or 'false'
LiteralDuration	<p>The format for Duration is as follows: ddddddhhmmss.ffffff:000</p> <p>The meaning of each field is as follows:</p> <ul style="list-style-type: none"> • dddddd is the number of days. • hh is the remaining number of hours. • mm is the remaining number of minutes. • ss is the remaining number of seconds. • fffffff is the remaining number of microseconds. • : (colon) indicates that the value is a duration. • 000 (the UTC offset field) is always zero for interval properties. <p>For example, a duration of 1 day, 13 hours, 23 minutes, 12 seconds, and 0 microseconds would be represented as follows: 0000001132312.000000:000</p> <p>The field values shall be zero-padded so that the entire string is always 25 characters in length. Fields that are not significant shall be replaced with the asterisk (*) character. Fields that are not significant are beyond the resolution of the data source. These fields indicate the precision of the value and can be used only for an adjacent set of fields, starting with the least significant field (mmmmmm) and continuing to more significant fields. The granularity for asterisks is always the entire field, except for the mmmmmm field, for which the granularity is single digits. The UTC offset field shall not contain asterisks. For example, if a duration of 1 day, 13 hours, 23 minutes, 12 seconds, and 125 milliseconds is measured with a precision of 1 millisecond, the format is: 0000001132312.125***:000.</p>
LiteralNull	Represents the state of having no value.

Specialization	Interpretation
LiteralOctetString	<p>A string of binary data of a specified length.</p> <p>The first four pairs of hexadecimal digits of the string value shall represent a length field, and any subsequent pairs shall represent the octets in the octet string. The four pairs of hexadecimal digits in the length field shall be interpreted as a 32-bit unsigned number where the first pair is the most significant byte. The number represented by the length field shall be the number of octets in the octet string plus four. For example, the empty octet string is represented as "0x00000004".</p>
LiteralReal	A literal bitfield contains a string of ones and zeros.
LiteralReference	Specifies the address of an instance as a URI.
LiteralSignedInteger	A literal representing a signed integer
LiteralString	<p>Implementations shall support a character repertoire for string typed values that is that defined by ISO/IEC 10646:2003 with its amendments ISO/IEC 10646:2003/Amd 1:2005 and ISO/IEC 10646:2003/Amd 2:2006 applied (this is the same character repertoire as defined by the Unicode Standard 5.0). It is recommended that implementations support the latest published UCS character repertoire in a timely manner.</p> <p>UCS characters in string typed values should be represented in Normalization Form C (NFC), as defined in The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms.</p> <p>UCS characters in string typed values shall be represented in a coded representation form that satisfies the requirements for the character repertoire stated in this clause. Other specifications are expected to specify additional rules on the usage of particular coded representation forms (see DSP0200 as an example). In order to minimize the need for any conversions between different coded representation forms, it is recommended that such other specifications mandate the UTF-8 coded representation form (defined in ISO/IEC 10646:2003).</p>

Specialization	Interpretation
LiteralTimeStamp	<p>A Value that is a LiteralTimestamp represents a date and time. A timestamp represents a point in time relative to the Universal Coordinated Time (UTC) on a specified date. The format for a timestamp string is: yyyyymmddhhmmss.mmmmmmsutc The meaning of each field is as follows:</p> <ul style="list-style-type: none"> • yyyy is a 4-digit year. • mm is the month within the year (starting with 01). • dd is the day within the month (starting with 01). • hh is the hour within the day (24-hour clock, starting with 00). • mm is the minute within the hour (starting with 00). • ss is the second within the minute (starting with 00). • mmmmmm is the microsecond within the second (starting with 000000). • s is a + (plus) or – (minus), indicating that the value is a timestamp with the sign of Universal Coordinated Time (UTC), which is basically the same as Greenwich Mean Time correction field. A + (plus) is used for time zones east of Greenwich, and a – (minus) is used for time zones west of Greenwich. • utc is the offset from UTC in minutes (using the sign indicated by s). <p>Timestamps are based on the proleptic Gregorian calendar, as defined in section 3.2.1, "The Gregorian calendar", of ISO 8601:2004. Because the string contains the time zone information, the original time zone can be reconstructed from the value. Therefore, the same point in time can be specified using different UTC offsets by adjusting the hour and minutes fields accordingly. For example, Monday, May 25, 1998, at 1:30:15 PM EST is represented as 19980525133015.0000000-300. An alternative representation of the same timestamp is 19980525183015.0000000+000. The field values shall be zero-padded so that the entire string is always 25 characters in length. Fields that are not significant shall be replaced with the asterisk (*) character. Fields that are not significant are beyond the resolution of the data source. These fields indicate the precision of the value and can be used only for an adjacent set of fields, starting with the least significant field (mmmmmm) and continuing to more significant fields. The granularity for asterisks is always the entire field, except for the mmmmmm field, for which the granularity is single digits. The UTC offset field shall not contain asterisks.</p>
LiteralUnlimitedNatural	Represents a, possibly unbounded, non-negative integer number representing an upper bound.
LimitedUnsignedInteger	Represents a non-negative integer number

1327

1328 **Generalizations**

- 1329
- CIMM::TypedElement (see 6.4.23)

1330 **Attributes**

- 1331 • No additional attributes

1332 **Associations**

- 1333 • No additional associations

1334 **Constraints**

- 1335 • No additional constraints

1336 **7 QualifierTypes**

1337 Within a metamodel conformant implementation, the set of supported metamodel extensions are
1338 specified via QualifierType declarations. This clause specifies DMTF defined QualifierTypes.

1339 A metamodel conformant implementation shall implement the Qualifiers for all scoped metamodel elements as
1340 specified by all included QualifierTypes.

1341 The set of DMTF defined QualifierTypes are subdivided into four categories, specified in the clauses
1342 below. They categories are:

- 1343 • Core metamodel extensions: Each QualifierType defines a metamodel extension that represents
1344 functionality common to most metamodels.
- 1345 • Specification based metamodel extensions: Each QualifierType defines a means to specify
1346 constraints on modeled elements.
- 1347 • Descriptive metamodel extensions: Each QualifierType defines a means to specify additional
1348 descriptive information about the modeled elements.

1349 **7.1 Core metamodel extensions**1350 **7.1.1 AggregationKind**

1351 AggregationKind that specifies the literals that define the kind of aggregation for a Property with type
1352 Reference. The aggregation may not be weakened. That is, a Qualifier on an overriding property may not
1353 demote a Property qualified as component to shared or bag, and may not demote a property qualified as
1354 shared to bag.

1355

Attribute	Value
scope	Property
type	CIMM::AggregationKind [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	CIMM::AggregationKind::none

1356

1357 **7.1.2 ArrayType**

1358 Indicates the type of array represented by the MethodReturn, Parameter or Property.

1359

Attribute	Value
scope	MethodReturn, Parameter, Property
type	CIMM::ArrayKind [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	CIMM::ArrayKind::bag

1360 **7.1.3 Description**

1361 The Description qualifier describes a named element.

1362

Attribute	Value
scope	Association, Class, Enumeration, EnumerationLiteral, Method, MethodReturn, Parameter, Property, QualifierType, Structure
type	CIMM::String [0..1]
policy	QualifierPolicyKind::applyMany
translatable	True
defaultValue	NULL

Comment [GME3]: Should we allow multiple Description entries

1363 **7.1.4 Max**

1364 The maximum number of elements referenced by a MethodReturn, Parameter, or Property with type
 1365 Reference. A null value implies that the upper value of the qualified element shall be unlimited ("**"). The
 1366 value of MAX modifies the upper bound of the multiplicity of referenced instances. The upper bound may
 1367 not be increased by an overriding MethodReturn, Parameter, or Property.

1368

Attribute	Value
Scope	MethodReturn, Parameter, Property
Type	CIMM::UnlimitedNatural [0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1369 **7.1.5 MaxLen**

1370 The value of MaxLen specifies the maximum length, in characters, of a string MethodReturn, Parameter,
 1371 Property or QualifierType. A null value implies that the maximum length is implementation dependent. The
 1372 value may not be increased by an overriding MethodReturn, Parameter, Property or QualifierType.

1373

Attribute	Value
scope	MethodReturn, Parameter, Property, QualifierType
type	CIMM::UnsignedInteger [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	NULL

1374 **7.1.6 MaxValue**

1375 The MaxValue qualifier specifies the maximum value of a numeric MethodReturn, Parameter, Property or
 1376 QualifierType. A null value implies that the maximum value is defined by the Type of the Property or
 1377 Parameter. The value may not be increased by an overriding MethodReturn, Parameter, Property or
 1378 QualifierType. The value shall not be greater than the maximum for the qualified element's Type.

1379

Attribute	Value
scope	MethodReturn, Parameter, Property, QualifierType
type	CIMM::UnsignedInteger [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	NULL

1380 **7.1.7 Min**

1381 The minimum number of elements referenced by a MethodReturn, Parameter, Property or QualifierType
 1382 with type Reference. A null value implies that the upper value of the qualified element shall be zero (0).
 1383 The value of MIN modifies the lower bound of the multiplicity of referenced instances. The lower bound
 1384 may not be decreased by an overriding MethodReturn, Parameter, Property or QualifierType.

1385

Attribute	Value
scope	MethodReturn, Parameter, Property, QualifierType
type	CIMM::UnsignedInteger [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	NULL

1386 **7.1.8 MinLen**

1387 The value of MinLen specifies the minimum length, in characters, of a string MethodReturn, Parameter,
 1388 Property or QualifierType. A null value implies that the minimum length is zero (0). The value may not be
 1389 decreased by an overriding MethodReturn, Parameter, Property or QualifierType.

1390

Attribute	Value
scope	MethodReturn, Parameter, Property, QualifierType
type	CIMM::UnsignedInteger [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false
defaultValue	NULL

1391 **7.1.9 MinValue**

1392 The MinValue qualifier specifies the minimum value of a numeric MethodReturn, Parameter, Property or
 1393 QualifierType. A null value implies that the maximum value is zero (0). The value may not be decreased
 1394 by an overriding MethodReturn, Parameter, Property or QualifierType. The value shall not be less than
 1395 the minimum for the qualified element's Type.

1396

Attribute	Value
scope	MethodReturn, Parameter, Property, QualifierType
type	CIMM::SignedInteger [0..1]
policy	QualifierPolicyKind::applyMany
translatable	false

defaultValue	NULL
--------------	------

1397 **7.1.10 NullOK (proposed)**

1398 The value of this qualifier indicates that the qualified MethodReturn, Property or Parameter may have no
 1399 value assigned to it. This is not the same as setting the lower bound of the element to zero (0). If the
 1400 MethodReturn, Property or Parameter has no value assigned, this shall be indicated by providing or
 1401 returning a literal NULL in place of a value that conforms to the Type of the qualified MethodReturn,
 1402 Property or Parameter.

1403

Attribute	Value
Scope	MethodReturn, Parameter, Property, QualifierType
Type	CIMM::Boolean [0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	false

1404 **7.1.11 Override**

1405 A non NULL value specifies an element (of the same name and type) defined in the ancestry of that
 1406 element. The qualified element takes the place of the specified element. The immediate descendents of
 1407 the qualifiedElement inherit the attributes of the qualifiedElement unless they also specify an Override
 1408 qualifier.

1409 An effective value of NULL (the default) indicates that the element is not overriding any element. If not
 1410 NULL, the value shall conform to the following formal syntax defined in ABNF:

```
1411 overriddenElementName = (typeName ".")* (
1412     methodName [ "." parameterName ] /
1413     (propertyName ".")* propertyName [ "::" enumerationLiteral ] )
```

1414 Where:

- 1415 • The typeName rule names the ancestor Type, (Association, Class, Enumeration, or Structure), that
 1416 owns the overridden element and is required if an element of the same name is exposed more than
 1417 once in the ancestry.
- 1418 • The methodName rule is required if the overridden element is a Method or by implication the
 1419 MethodReturn. If the overridden element is a Parameter, then it shall be specified.
- 1420 • The propertyName rule is required if a Property is overridden.

1421

Attribute	Value
Scope	Association, Class, Enumeration, EnumerationLiteral, Method, MethodReturn, Parameter, Property, Structure
Type	CIMM::String [0..1]

Policy	QualifierPolicyKind::restricted
Translatable	False
defaultValue	NULL

1422 **7.1.12 SchemaURI (proposed)**

1423 A non-NULL value is a fully qualified URI that identifies the schema that specifies the qualified element.

1424

Attribute	Value
Scope	Association, Class, Enumeration, Schema, Structure
Type	CIMM::String [0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	False
defaultValue	NULL

1425 **7.1.13 Read**

1426 The modeling semantics of a property support retrieving its value or values. The purpose of this qualifier
 1427 is to capture modeling semantics and not to address more dynamic characteristics such as provider
 1428 capability or authorization rights.

1429

Attribute	Value
Scope	Property
Type	CIMM::Boolean [0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	False
defaultValue	True

1430 **7.1.14 Write**

1431 The modeling semantics of a property support updating its value or values. The purpose of this qualifier
 1432 is to capture modeling semantics and not to address more dynamic characteristics such as provider
 1433 capability or authorization rights.

1434

Attribute	Value
Scope	Property
Type	CIMM::Boolean [0..1]
Policy	QualifierPolicyKind::applyMany

Translatable	False
defaultValue	False

1435 **7.1.15 Version**

1436 A non NULL value of specifies the version of a Schema or Type. The value should be incremented when
 1437 changes are made to that element, including changes to its constituent elements. Version is not
 1438 inherited. In other words, the version change of a super type does not require the version in the sub type
 1439 to be updated.

1440 The string representing the version comprises three decimal integers separated by periods; that is,
 1441 M.N.U, as defined by the following ABNF:

1442 `versionFormat = decimalValue "." decimalValue "." decimalValue`

1443 The meaning of M.N.U is as follows:

- 1444 **M** – The major version in numeric form of the change to the class.
- 1445 **N** – The minor version in numeric form of the change to the class.
- 1446 **U** – The update (for example, errata, patch, ...) in numeric form of the change to the class.

1447 EXAMPLES:

1448 `Version("2.7.0")`

1449
 1450 `Version("1.0.1")`

1451

Attribute	Value
Scope	Association, Class, Enumeration, Schema, Structure
Type	CIMM::String[0..1]
Policy	QualifierPolicyKind::restricted
Translatable	False
defaultValue	NULL

1452 **7.2 Specification based metamodel extensions**

1453 **7.2.1 Correlatable**

1454 The values of the Correlatable qualifier is used to specify the role a property plays within a set of
 1455 properties that taken together identify a particular resource.

1456 The value of each entry in the Correlatable qualifier entry shall identify a role according to the following
 1457 ABNF syntax:

1458 `roleID = org_name ":" set_name ":" role_name`

- 1459 • `org_name`: shall be a copyrighted, trademarked or otherwise unique name that is owned by the
 1460 business entity defining `set_name`, or is a registered ID that is assigned to the business entity by a
 1461 recognized global authority. `org_name` shall not contain a colon (":"). For DMTF defined roleID
 1462 values, the `org_name` shall be "CIM".

1463 • set_name: shall be unique within the context of org_name and identifies a specific set of correlatable
1464 properties. set_name shall not contain a colon (":").

1465 • role_name: shall be unique within the context of org_name and set_name and identifies the
1466 semantics or role that the property plays within the Correlatable comparison.

1467 Two instances match if they each contain a set of properties such that for some org_name and set_name,
1468 each instance has a property with the same value and same role_name.

1469 roleID values shall be compared case-insensitively. For example, these are considered to match:

1470 "Acme:Set1:Role1" and "ACME:set1:role1"

1471

Attribute	Value
Scope	Property
Type	CIMM::String[0..*]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1472 7.2.2 MappingStrings

1473 The value of each entry specifies a corresponding representation for the qualified element in another
1474 standard. See Annex G for standard mapping formats.

1475

Attribute	Value
Scope	Association, Class, Enumeration, EnumerationLiteral, Property, Parameter, Method, MethodReturn, Structure
Type	CIMM::String[0..*]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1476 7.2.3 ModelCorrespondence

1477 Each value names an element of this or another schema. The semantics indicate a correspondence
1478 between the qualified element and the named element. That correspondence should be described in the
1479 definition of those elements, but may be described elsewhere.

1480 The format of each value is specified by the following ABNF:

1481 correspondingElementName =

1482 `[(schemaURI "/" / (schemaName) "/")* (typeName ".")*] /`

1483 `(typeName ".")*`

1484 (methodName ["." parameterName] /
 1485 (propertyName ".")* propertyName ["::" enumerationLiteral])

1486 Where:

- 1487 • The schemaURI rule is required if the corresponding Method or Property is defined in a different
 1488 Schema.
- 1489 • The typeName rule names the ancestor Type, (Association, Class, Enumeration, or Structure), that
 1490 owns the corresponding element and is required if an element of the same name is exposed more
 1491 than once in the ancestry.
- 1492 • The methodName rule is required if the overridden element is a Method or by implication the
 1493 MethodReturn. If the overridden element is a Parameter, then it shall be specified.
- 1494 • The propertyName rule is required if a Property is overridden.

1495 The basic relationship between the referenced elements is a "loose" correspondence, which simply
 1496 indicates that the elements are coupled. This coupling may be unidirectional. Additional qualifiers may be
 1497 used to describe a tighter coupling.

1498 The following list provides examples of several correspondences found in CIM and vendor schemas:

- 1499 • A property provides more information for another. For example, an enumeration has an allowed
 1500 value of "Other", and another property further clarifies the intended meaning of "Other." In another
 1501 case, a property specifies status and another property provides human-readable strings (using an
 1502 array construct) expanding on this status. In these cases, ModelCorrespondence is found on both
 1503 properties, each referencing the other.
- 1504 • A property is defined in a subclass to supplement the meaning of an inherited property. In this case,
 1505 the ModelCorrespondence is found only on the construct in the subclass.
- 1506 • Multiple properties taken together are needed for complete semantics. For example, one property
 1507 may define units, another property may define a multiplier, and another property may define a
 1508 specific value. In this case, ModelCorrespondence is found on all related properties, each
 1509 referencing all the others.
- 1510 • Multi-dimensional arrays are desired. For example, one array may define names while another
 1511 defines the name formats. In this case, the arrays are each defined with the ModelCorrespondence
 1512 qualifier, referencing the other array properties or parameters. Also, they are indexed and they carry
 1513 the ArrayType qualifier with the value "Indexed."

1514 The semantics of the correspondence are based on the elements themselves. ModelCorrespondence is
 1515 only a hint or indicator of a relationship between the elements.

1516

Attribute	Value
Scope	Association, Class, Enumeration, EnumerationLiteral, Property, Parameter, Method, MethodReturn, Structure
Type	CIMM::String[0..*]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1517 **7.2.4 OCLConstraint**

1518 Each value of an OCLConstraint entry specifies an OMG Object Constraint Language (OCL) statement
 1519 that defines a definition, invariant, derivation, initialization, precondition, postcondition, or bodycondition
 1520 constraint, (see the [Object Constraint Language](#) specification).

1521 The OCL context of these constraints (that is, what "self" in OCL refers to) is the Class Instance
 1522 containing the qualified element.

1523 Initialization constraints specified for a Property specify constraints on the initial value of a Property. Note
 1524 that such constraints do not provide a value. If both a default value and initialization constraints are
 1525 specified, they shall be consistent with each other.

1526 Other specifications may define additional means to determine the initial value of a Property; for example,
 1527 management profiles may define initialization constraints, or operation specifications may define that
 1528 operations that cause new instances to come into existence support the ability to redefine the schema
 1529 defined initialization constraints.

1530 Unless overridden, default values and constraints on an overridden Property propagate to the overriding
 1531 Property.

- 1532 • An optional Constraint on the result values of an invocation of this Method.

1533 *bodyCondition: Constraint[0..1]*

- 1534 • An optional set of Constraints specifying the state of the system when the Method is completed.

1535 *postcondition: Constraint[0..*]*

- 1536 • An optional set of Constraints on the state of the system when the Method is invoked.

1537 *precondition: Constraint[0..*]*

- 1538 • A bodyCondition can only be specified for a query Method.

1539 *bodyCondition->notEmpty() implies isQuery*

- 1540 • A collection of Constraints defined in this NamingContext.

1541 *ownedRule : Constraint [0..*]*

- 1542 • Specifies whether the values of this Property can be computed from other information

1543 *derived : Boolean [1] = false*

1544 See clause 10 for more information on the specification of OCL constraints.

1545

Attribute	Value
Scope	Association, Class, Enumeration, EnumerationLiteral, Property, Parameter, QualifierType, Method, MethodReturn, Structure
Type	CIMM::String[0..*]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1546 **7.3 Descriptive metamodel extensions**

1547 **7.3.1 Counter**

1548 A value of true indicates that an unsigned element has a value (or values) that monotonically increase
 1549 until a maximum value of $2^n - 1$ is reached, then starts increasing again from zero (0), where n is the
 1550 width of the element's type, (i.e. 8, 16, 32, 64, or 128).

1551

Attribute	Value
Scope	MethodReturn, Parameter, Property, QualifierType
Type	CIMM:: Boolean[0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1552 **7.3.2 Deprecated**

1553 A non NULL value indicates that support for the qualified element is planned to be removed in the next
 1554 major version of the schema that defines the element. This is informational only and does not affect the
 1555 qualified element's semantics or support requirements.

1556 Zero, one, or a group of replacement elements may be specified. If there is no replacement element,
 1557 then the qualifier shall contain a single entry of "No value".

1558 Replacement elements shall be specified using the syntax defined in the following ABNF:

1559

```
replacement = ("No value" /
              (typeName "." typeName)*
              [ "." methodName [ "." parameterName ] /
              "." (propertyName ".")* propertyName [ ":" enumerationLiteral ] ] )
```

1563

Where:

- 1564 • The typeName rule names the ancestor Type, (Association, Class, Enumeration, or Structure), that
 1565 owns the replacement element.
- 1566 • The methodName rule is required if the replaced element is a Method or by implication the
 1567 MethodReturn. If the overridden element is a Parameter, then it shall be specified.
- 1568 • The propertyName rule is required if a Property is replaced.

1569 Existing implementations should additionally add support for replacement elements if they are specified.

1570 New implementations should not use deprecated elements.

1571

Attribute	Value
Scope	Association, Class, Enumeration, EnumerationLiteral, Property, Parameter, Method, MethodReturn, Structure

Type	CIMM::String[0..*]
Policy	QualifierPolicyKind::restricted
Translatable	false
defaultValue	NULL

1572 **7.3.3 DisplayName**

1573 Provides a name for display on a user interface instead of the actual name of the element.

1574

Attribute	Value
scope	Association, Class, Enumeration, EnumerationLiteral, Method, MethodReturn, Parameter, Property, QualifierType, Structure
type	CIMM::String [0..1]
policy	QualifierPolicyKind::applyMany
translatable	True
defaultValue	NULL

1575 **7.3.4 Experimental**

1576 If true, the qualified element has 'experimental' status. The implications of experimental status are
1577 specified by the schema owner.

1578 If false, the qualified element has 'final' status. Elements with 'final' status may not be modified in
1579 backwards incompatible ways within a major schema version, (see Annex B.)

1580 In a DMTF-produced schema, experimental elements are subject to change and are not included in 'final'
1581 schema. Elements with 'experimental' status may be modified in backwards incompatible ways within a
1582 major schema version.

1583 Experimental elements are published for developing implementation experience. Based on
1584 implementation experience: changes may occur to this element in future releases; the element may be
1585 standardized "as is"; or the element may be removed.

1586 An implementation does not have to support an experimental element to be compliant to a published
1587 DMTF 'final' schema.

1588 When applied to a class, the Experimental qualifier conveys experimental status to the class itself, as well
1589 as to all sub elements of that class. Therefore, if a class already bears the Experimental qualifier, it is
1590 unnecessary also to apply the Experimental qualifier to any of its properties or features, and such
1591 redundant use is discouraged.

1592 No element shall be qualified as both experimental and deprecated.

1593 Experimental elements for which a decision is made to not eventually take them final should be removed
1594 from their schema.

1595 NOTE 1: The addition or removal of the Experimental qualifier does not require the version information to be
1596 updated.

1597

Attribute	Value
scope	Association, Class, Enumeration, EnumerationLiteral, Method, MethodReturn, Parameter, Property, QualifierType, Structure
type	CIMM::Boolean[0..1]
policy	QualifierPolicyKind::restricted
translatable	True
defaultValue	false

1598 **7.3.5 Guage**

1599 A value of true indicates that an integer element has a value (or values) that that may increase or
1600 decrease in any order of magnitude.

1601 The value of a gauge is capped at the limits of the element's Type. If the information being modeled
1602 exceeds an implied limit, the value represented is that limit. Values do not wrap. For unsigned integers,
1603 the limits are zero (0) to 2^n-1 , inclusive. For signed integers, the limits are $-(2^{n-1})$ to
1604 $2^{n-1}-1$, inclusive. Where n is 8, 16, 32, 64, or 128 depending on the Type of the qualified element.

1605

Attribute	Value
Scope	MethodReturn, Parameter, Property, QualifierType
Type	CIMM:: Boolean[0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	false

1606 **7.3.6 IsPUnit**

1607 The qualified string represents a programmatic unit of measure. The value of the string element follows
1608 the syntax for programmatic units, as defined in Annex F.

1609

Attribute	Value
Scope	Parameter, Property
Type	CIMM:: Boolean[0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1610 **7.3.7 ResourceDescription (proposed)**

1611 Provides a description of the underlying resource that the model element is representing.

1612

Attribute	Value
scope	Association, Class, Enumeration, EnumerationLiteral, Method, MethodReturn, Parameter, Property, QualifierType, Structure
type	CIMM::String [0..1]
policy	QualifierPolicyKind::applyMany
translatable	True
defaultValue	NULL

1613 **7.3.8 PUnit**

1614 A non NULL value indicates the programmatic unit of measure of a numeric element.

1615 Note: String typed schema elements that are used to represent numeric values in a string format cannot have the
 1616 PUnit qualifier specified, since the reason for using string typed elements to represent numeric values is typically
 1617 that the type of value changes over time, and hence a programmatic unit for the element needs to be able to
 1618 change along with the type of value. This can be achieved with a companion schema element whose value
 1619 specifies the programmatic unit in case the first schema element holds a numeric value. This companion schema
 1620 element would be string typed and the IsPUnit qualifier be set to true.

1621 The syntax for programmatic units is defined in Annex F.

1622

Attribute	Value
Scope	MethodReturn, Parameter, Property, QualifierType
Type	CIMM::String[0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	false
defaultValue	NULL

1623 **7.3.9 XMLNamespaceName**1624 A non NULL value of the qualifier shall be the URI of an XML schema that defines the format of the XML
1625 instance document that is the value of the qualified string element.

1626 As defined in NamingContexts in XML, the format of the XML Namespace name shall be that of a URI
 1627 reference as defined in [RFC3986](#). Two such URI references may be equivalent even if they are not equal
 1628 according to a character-by-character comparison (e.g., due to usage of URI escape characters or
 1629 different lexical case).

1630 If the value of the XMLNamespaceName qualifier overrides a non-NULL qualifier value specified on an
 1631 ancestor of the qualified element, the XML schema specified on the qualified element shall be a subset or
 1632 restriction of the XML schema specified on the ancestor element, such that any XML instance document

DSP0004

Common Information Model (CIM) Metamodel

1633 that conforms to the XML schema specified on the qualified element also conforms to the XML schema
1634 specified on the ancestor element.

1635 No particular XML schema description language (e.g., W3C XML Schema as defined in [XML Schema](#)
1636 [Part 0: Primer Second Edition](#) or RELAX NG as defined in [ISO/IEC 19757-2:2008](#)) is implied by usage of
1637 this qualifier.

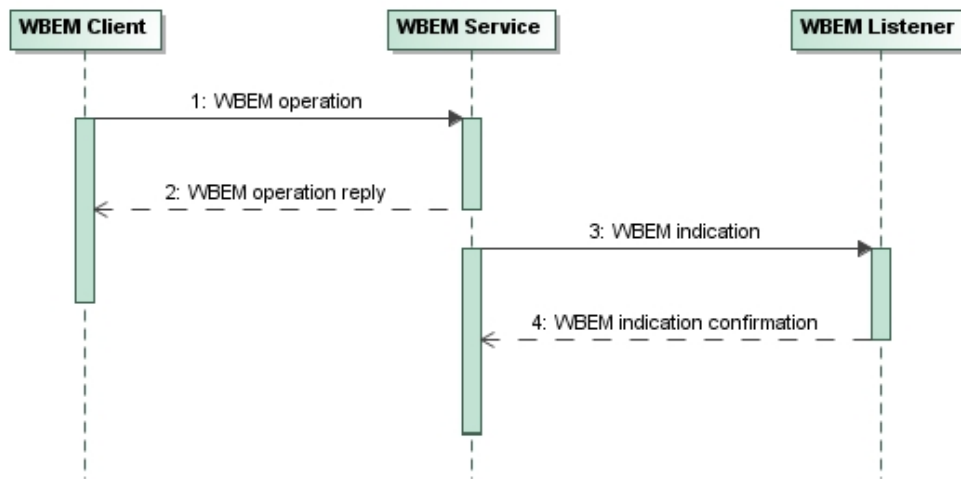
1638

Attribute	Value
Scope	Association, Class, Enumeration, Schema, Structure
Type	CIMM::String [0..1]
Policy	QualifierPolicyKind::applyMany
Translatable	False
defaultValue	NULL

1639 **8 Model element naming**

1640 The CIM metamodel facilitates sharing of management information among a variety of management
 1641 service implementations by requiring a naming strategy that enables model elements holding
 1642 management information to be identified and addressed.

1643 Figure 2 shows the DMTF Web Based Enterprise Management (WBEM) environment in which model
 1644 elements are known and addressed.



1645

1646 **Figure 2: WBEM Infrastructure**

1647 The primary requirement is that names can be used to.

1648 The naming mechanism addresses the following requirements:

- 1649 • Unambiguous and representation independent identification of model elements accessed from a
 1650 WBEM service.
- 1651 • Ability to determine when two object names reference the same CIM object. This entails
 1652 location transparency so that there is no need for a consumer of an object name to understand
 1653 which management platforms proxy the instrumentation of other platforms.

1654 In each of two environments, there is a natural hierarchy of naming for model elements. The environments
 1655 are a Schema realized from CIMM::Schema and a WBEM Service.

1656 This clause defines naming in these two environments.

1657 **8.1 Model element name syntax**

1658 Each Schema has a location independent URI that identifies it. Within the context of a Schema, model
 1659 elements are named as follows:

- 1660 1) Top level Schema is addressed via an RFC3986 conformant URI
 1661 *topSchemaURI = URI*
- 1662 2) component Schema elements
 1663 *schemaURI = topSchemaURI ("::" schemaName)**
- 1664 3) Property Rule
 1665 *propertyRule = propertyName ("." propertyName)* ["::" enumerationLiteralName]*
- 1666 4) Parameter Rule
 1667 *parameterRule = parameterName ("." propertyName)* ["::" enumerationLiteralName]*
- 1668 5) Method Rule
 1669 *methodRule = methodName ["." ParameterRuleName]*
- 1670 6) Structure
 1671 *structure = structureName ["." propertyRule]*
- 1672 7) Enumeration
 1673 *enumerationPath = enumerationName ["::" enumerationLiteralName]*
- 1674 8) Association or Class
 1675 *class = className*
 1676 *("." (propertyRule / methodRule) /*
 1677 *("::" (structure / enumeration))*
- 1678 9) Instance Name
 1679 *instanceName = keyValue*
- 1680 10) Instance Declaration Path
 1681 *instance = className ["::" instanceName ["." propertyRule]*
- 1682 11) Model Declaration Path
 1683 *schemaModelURI = schemaURI "::
 1684 (class / structure / enumeration / instance / qualifierTypeName)*

Comment [GME4]: CIMv2 codes this as ". (propertyName "=" value)+ CIM v3 only has one key, so the propertyName is not needed. Using "::" to distinguish.

Comment [GME5]: CIMv2 codes this as ". (propertyName "=" value)+ CIM v3 only has one key, so the propertyName is not needed. Using "::" to distinguish.

1685 Each instance of a WBEM Service has one or more URLs that addresses it. This clause defines how
 1686 model elements are accessed in the context of an addressed WBEM service instance.

- 1687 12) The instance of a WBEM Service is addressed by one or more RFC3986 conformants URLs.
 1688 *namespacePath = URL*
- 1689 13) Schema declarations known to the WBEM service
 1690 *serviceSchemaPath = namespacePath "::
 1691 14) Model Elements known to the WBEM service
 1692 *serviceModelPath = serviceSchemaPath "::
 1693 (class / structure / enumeration / instance / qualifierTypeName)**

Comment [GME6]: I'd prefer this to be "::
 However a single "." is consistent with CIM v2.

Comment [GME7]: I'd prefer this to be "::
 However, the "_" is consistent with CIM v2.

Comment [GME8]: This proposal allows a QualifierTypeName to be qualified by a schemaName. CIM v2 doesn't qualify DMTF defined q_ualifierTypeNames with a schemaName.

1694 **8.2 Matching model element names**

1695 Matching of model element names (which consist of UCS characters) as defined in this document shall
 1696 be performed as if the following algorithm was applied:

1697 Any lower case UCS characters in the CIM names are translated to upper case.

1698 The CIM names are considered to match if the string identity matching rules defined in chapter 4 "String
 1699 Identity Matching" of [Character Model for the World Wide Web 1.0: Normalization](#) match when applied to
 1700 the upper case CIM names.

1701 In order to eliminate the costly processing involved in this, specifications may define simplified processing
1702 for applying this algorithm. One way to achieve this is to mandate that Normalization Form C (NFC),
1703 defined in [The Unicode Standard, Version 5.2.0, Annex #15: Unicode Normalization Forms](#), which allows
1704 the normalization to be skipped when comparing the names.

1705 **8.3 Identity of CIM objects**

1706 The same model element is addressed by two paths if the class, structure, enumeration, instance, or
1707 QualifierType component of the path is identical and the respective serviceSchemaPath matches. Since
1708 this depends on whether their namespace paths match, it may not be possible to determine this directly.

1709 Two different CIM objects (e.g., instances) can represent aspects of the same managed object. In other
1710 words, identity at the level of CIM objects is separate from identity at the level of the represented
1711 managed objects.

1712 **9 Supported schema modifications**

1713 This clause lists typical modifications of schema definitions and qualifier type declarations and defines
1714 their compatibility.

1715 Table 5 lists modifications of an existing schema definition (including an empty schema). The compatibility
1716 of the modification is indicated for WBEM clients that utilize the modified element, and for a WBEM
1717 service that implements the modified element. Compatibility for WBEM service that utilizes the modified
1718 element (e.g., via so called "up-calls") is the same as for a WBEM client that utilizes the modified element.

1719 The compatibility for WBEM clients as expressed in Table 5 assumes that the WBEM service remains
1720 unchanged and is exposed to a WBEM service that was updated to fully reflect the schema modification.

1721 The compatibility for WBEM clients as expressed in Table 5 assumes that the WBEM client remains
1722 unchanged but is exposed to the modified schema that is loaded into the WBEM service.

1723 Compatibility is stated as follows:

- 1724 • Transparent – the respective component does not need to be changed in order to properly deal
1725 with the modification
- 1726 • Not transparent – the respective component needs to be changed in order to properly deal with
1727 the modification

1728 Schema modifications qualified as transparent for both WBEM clients and WBEM services are allowed in
1729 a minor version update of the schema. Any other schema modifications are allowed only in a major
1730 version update of the schema.

1731 The schema modifications listed in Table 5 cover simple cases, which may be combined to yield more
1732 complex cases. For example, a typical schema change is to move existing properties or methods into a
1733 new superclass. The compatibility of this complex schema modification can be determined by
1734 concatenating simple schema modifications listed in Table 5, as follows:

1735 1) SM1: Adding a class to the schema:

1736 The new superclass gets added as an empty class with (yet) no superclass

1737 2) SM3: Inserting an existing class that defines no properties or methods into an inheritance
1738 hierarchy of existing classes:

1739 The new superclass gets inserted into an inheritance hierarchy

1740 3) SM8: Moving an existing property from a class to one of its superclasses (zero or more times)

1741 Properties get moved to the newly inserted superclass

1742 4) SM12: Moving a method from a class to one of its superclasses (zero or more times)

1743 Methods get moved to the newly inserted superclass

1744 The resulting compatibility of this complex schema modification for WBEM clients is transparent, since all
 1745 these schema modifications are transparent. Similarly, the resulting compatibility for WBEM services is
 1746 transparent for the same reason.

1747 Some schema modifications cause other changes in the schema to happen. For example, the removal of
 1748 a class causes any associations or method parameters that reference that class to be updated in some
 1749 way.

1750 **Table 5: Compatibility of Schema Modifications**

Schema Modification	Compatibility for clients	Compatibility for 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 clients that examine classes are prepared to deal with new classes in the schema and with new classes 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 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

Schema Modification	Compatibility for clients	Compatibility for servers	Allowed in a Minor Version Update of the Schema
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 clients are prepared to deal with any new properties in classes and instances.	Transparent If the 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) server will not include the new property in any instances of the class, and a client that knows about the new property will interpret it as having the NULL value.	Yes
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 clients that examine classes are prepared to deal with such moved properties. For 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 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

Schema Modification	Compatibility for clients	Compatibility for servers	Allowed in a Minor Version Update of the Schema
SM10: Adding a method to an existing class that is not overriding a method	Transparent It is assumed that any clients that examine classes are prepared to deal with such added methods.	Transparent It is assumed that a server is prepared to return an error to clients indicating that the added method is not implemented.	Yes
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 clients that examine classes are prepared to deal with such moved methods. For 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 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
SM17: Changing the class referenced by a reference in an association to a subclass of the previously referenced class	Transparent	Not Transparent	No

Schema Modification	Compatibility for clients	Compatibility for servers	Allowed in a Minor Version Update of the Schema
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
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

Schema Modification	Compatibility for clients	Compatibility for servers	Allowed in a Minor Version Update of the Schema
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 0	As defined in the qualifier description in 0	Yes, if transparent for both clients and servers, otherwise No

1751 5) Factory approach to populate the properties of any instances to be returned:
 1752 Some server architectures (e.g., CMPI-based CIM providers) support factory methods that
 1753 create an internal representation of a CIM instance by inspecting the class object and creating
 1754 property values for all properties exposed by the class and setting those values to their class
 1755 defined default values. This delegates the knowledge about newly added properties to the
 1756 schema definition of the class and will return instances that are compliant to the modified
 1757 schema without changing the code of the server. A subsequent release of the server can then
 1758 start supporting the new property with more reasonable values than the class defined default
 1759 value.

1760 Table 6 lists modifications of qualifier types. The compatibility of the modification is indicated for an
 1761 existing schema. Compatibility for clients or servers is determined by Table 6 (in any modifications that
 1762 are related to qualifier values).

1763 The compatibility for a schema as expressed in Table 6 assumes that the schema remains unchanged
 1764 but is confronted with a qualifier type declaration that reflects the modification.

1765 Compatibility is stated as follows:

- 1766 • Transparent – the schema does not need to be changed in order to properly deal with the
 1767 modification
- 1768 • Not transparent – the schema needs to be changed in order to properly deal with the
 1769 modification

1770 CIM supports extension schemas, so the actual usage of qualifiers in such schemas is by definition
 1771 unknown and any possible usage needs to be assumed for compatibility considerations.

1772 **Table 6: 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

Qualifier Type Modification	Compatibility for Existing Schema	Allowed in a Minor Version Update of the Schema
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

1773 10 Object constraint language

1774 This clause specifies the subset of the OCL language used in this specification, see the [OCL](#)
 1775 [Specification](#). In addition, some OCL query functions are used as defined in the [UML Superclass](#)
 1776 [Specification](#).

1777 NOTE 1: All OCL constraints are specified on the element that provides the context for the constraint. This
 1778 context is represented in OCL by the keyword "self". In most expressions, "self" does not need to be explicitly
 1779 stated. For example, if a constraint element contains a property P, then self.P=0 and P=0 are equivalent.

1780 NOTE 2: The OCL language allows traversing associations in both directions, regardless of whether or not the
 1781 association ends are owned by the association or the associated class. However, OCL engines may only support
 1782 traversal from association ends owned by the associated UML metaclasses, but not from ends owned by the
 1783 associations themselves. The OCL in any constraints defined in this document accommodates for such OCL
 1784 engines in that it only traverses association ends owned by the associated UML metaclasses.

1785 The Object Constraint Language (OCL) is a formal language to describe expressions on models. It is
 1786 defined by the Open Management Group (OMG) in the [Object Constraint Language](#) specification, which
 1787 describes that OCL is intended as a specification language. That is, its statements are intended to be
 1788 evaluated at design time, not run time.

1789 OCL expressions do not change anything in a model, but rather are intended to evaluate whether or not a
 1790 modeled system is conformant to a specification. This means that the state of the system will never
 1791 change because of the evaluation of an OCL expression.

1792 10.1 Navigation across associations

1793 Associations in the the CIM metamodel own the references to the associated classes. This means that
 1794 the reference property is not defined in the NamingContext of the referencing class, but rather in the
 1795 NamingContext of the association class. Because of this, models developed using associations may
 1796 have multiple associations with reference properties having the same name.

1797 The metamodel alternatively allows a property in one class to reference another.

1798 Starting at one class in a model, typical OCL navigation simply follows a referencing property to an
 1799 associated class. However when associations are used, it is possible (and in the CIM schema somewhat
 1800 likely) that simply following the role names will be ambiguous. This ambiguity is resolved by first
 1801 referencing the association class name, and then following the role from there. This strategy is described
 1802 in the [Object Constraint Language](#) specification in its sections 7.5.4 "Navigation to Association Classes"
 1803 and 7.5.5 "Navigation from Association Classes".

1804 10.2 Self

1805 Each OCL statement is made in the context of an instance. The reserved word self is used to refer that
 1806 instance. For example, in the metamodel, if the context is the Property metaclass, then self refers to an
 1807 instance of the Property metaclass. Within a model that is conformant to the CIM metamodel, self will
 1808 always refer to either a class or an association instance.

1809 10.3 Types

1810 The following types used in this document:

- 1811 • Boolean, Integer, and String.
 - 1812 • Boolean values are "true" and "false"
 - 1813 • Strings are alternatively treated as a sequence of characters indexed from 1 to size().
- 1814 • The classes that define the CIM metamodel
- 1815 • Collections, Sets, and Sequences
- 1816 • Enumerations

1817 10.4 Operations and precedence

1818 Table 7 lists the operations in order of precedence.

1819 **Table 7: Operations**

Operator	
"(" and ")"	Encapsulate and deencapsulate operations. All operations within an encapsulation are evaluated before any values outside of an encapsulation.
"." and "->"	Dot operations take the value, and arrow operations dereference
"not" and "-"	Logical not and arithmetic negative operations
"*" and "/"	Multiplication and division operations
"+" and "-"	Addition and Subtraction operations
"if-then-else-endif"	

"<", ">", "<=", ">="	Comparison operations
"=", "<>"	Equality operations
"and"	Logical boolean conjunction operation
"or"	Logical boolean disjunction operation
"xor"	Logical boolean exclusive disjunction (exclusive or) operation
"implies"	If this is true, then this other thing must be true
"let-in"	

1820 **10.4.1 OCL expression keywords**

1821 The following are OCL reserved words.

1822 **Table 8: OCL Expression keywords**

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

1823 **10.4.2 OCL operations**

1824 Table 9 lists OCL operations used by this specification.

1825 **Table 9: OCL Operations**

Operation	Result
asSet()	Converts a Bag or Sequence to a Set (duplicates are removed.)
at()	The i^{th} element of an Ordered Set or a Sequence. (Note: A String is treated as a Sequence of characters.)
collect()	A derived collection of elements
concat()	The specified string appended to the end of self.
count()	The number of times a specified object occurs in self
excludes()	True if the specified object is not an element of self
excluding()	The set containing all the elements of self, but without the specified object
exists()	True if the expression evaluates to true for at least one element in a source collection.
forAll()	True if the expression evaluates to true for every element in a source collection.
includes()	True if the specified object is an element of self
includesAll()	True if self contain all of the elements in the specified collection

isEmpty()	True if self is the empty collection
notEmpty()	True if self is not the empty collection
oclAsType()	Casts self to the specified type if it is in the hierarchy of self or undefined.
oclIsKindOf()	True if self is a kind of the specified type.
oclIsUndefined()	True if the result is undefined or Null.
select()	The subset of elements from the a source collection for which the expression evaluates to true.
size()	The number of elements in the collection of self
substring()	The substring of self starting at a first character number and including all characters up to a second character number. Character numbers run from 1 to self.size().
toUpperCase()	Converts self to upper case, if appropriate to the locale. Otherwise, returns the same string as self.
union()	The union of self and a bag

1826 10.5 OCL statements

1827 The following sub clauses define the subset of OCL used by this document. Other elements of OCL may
1828 be used by vendors or others that extend the metamodel defined by this document.

1829 By default, ABNF rules (including literals) are to be assembled without inserting any additional whitespace
1830 characters, consistent with [RFC5234](#). If an ABNF rule states "whitespace allowed", zero or more of the
1831 following whitespace characters are allowed between any ABNF rules (including literals) that are to be
1832 assembled:

- 1833 • U+0009 (horizontal tab)
- 1834 • U+000A (linefeed, newline)
- 1835 • U+000C (form feed)
- 1836 • U+000D (carriage return)
- 1837 • U+0020 (space)

1838 10.5.1 Comment statement

1839 Comments in OCL are written using either of two techniques:

- 1840 The line comment starts with the string '--' and ends with the next newline.
- 1841 The paragraph comment starts with the string '/*' and ends with the string '*/.' Paragraph
1842 comments may be nested.

1843 10.5.2 Let expressions

1844 The let expression allows a variable to be defined and used multiple times within an OCL constraint.

1845 `let_expression = "let" varName ":" typeName "=" varInitializer in ocl_statement`

- 1846 • varName is a name for a variable
- 1847 • typeName is the OCL type of the variable

- 1848
- varInitializaer is the OCL statement that evaluates to a typeName conformant value
- 1849
- ocl_statement is an OCL statement that utilizes varName

1850 10.5.3 OCL definition constraints

1851 OCL definition constraints define OCL attributes and OCL operations that are reusable by other OCL
1852 constraints in the same OCL context.

1853 The attributes and operations defined by OCL definition constraints shall be visible for:

- 1854
- OCL definition and invariant constraints defined in subsequent entries of the the same
1855 ClassConstraint
 - OCL constraints of MethodConstraint or PropertyConstraint entries in this class, (or association), and
1856 any of its subclasses.
1857

1858 A value specifying an OCL definition constraint shall conform to the following formal syntax defined in
1859 ABNF (whitespace allowed):

1860 `ocl_definition = "def" [ocl_name] ":" ocl_statement`

- 1861
- ocl_name is a name by which the defined attribute or operation can be referenced.
- 1862
- ocl_statement is the OCL statement of the definition constraint, which defines the reusable attribute
1863 or operation.
- 1864
- Note: The use of the OCL keyword *self* to scope a reference to a property is optional.

1865 10.5.4 OCL invariant constraints

1866 OCL invariant constraints specify a boolean expression that shall be true for the lifetime of an instance of
1867 the qualified class or association.

1868 A value specifying an OCL definition invariant constraint shall conform to the following formal syntax
1869 defined in ABNF (whitespace allowed):

1870 `ocl_invariant = "inv" [ocl_name] ":" ocl_statement`

- 1871
- ocl_name is a name by which the invariant expression can be referenced.
- 1872
- ocl_statement is the OCL statement of the invariant constraint, which defines the boolean
1873 expression.
- 1874
- Note: The use of the OCL keyword *self* to scope a reference to a property is optional.

1875 10.5.5 OCL precondition constraint

1876 An OCL precondition constraint is expressed as a typed OCL expression that specifies whether the
1877 precondition is satisfied. The type of the expression shall be boolean. For the method to complete
1878 successfully, all preconditions of a method shall be satisfied before it is invoked.

1879 A string value specifying an OCL precondition constraint shall conform to the formal syntax defined in
1880 ABNF (whitespace allowed):

1881 `ocl_precondition_string = "pre" [ocl_name] ":" ocl_statement`

1882 Where:

1883 ocl_name is the name of the OCL constraint.

1884 ocl_statement is the OCL statement of the precondition constraint, which defines the boolean expression.

1885 10.5.6 OCL postcondition constraint

1886 An OCL postcondition constraint is expressed as a typed OCL expression that specifies whether the
 1887 postcondition is satisfied. The type of the expression shall be boolean. All postconditions of the method
 1888 shall be satisfied immediately after successful completion of the method.

1889 A string value specifying an OCL post-condition constraint shall conform to the following formal syntax
 1890 defined in ABNF (whitespace allowed):

1891 `ocl_postcondition_string = "post" [ocl_name] ":" ocl_statement`

1892 Where:

1893 `ocl_name` is the name of the OCL constraint.

1894 `ocl_statement` is the OCL statement of the post-condition constraint, which defines the boolean
 1895 expression.

1896 10.5.7 OCL body constraint

1897 An OCL body constraint is expressed as a typed OCL expression that specifies the return value of a
 1898 method. The type of the expression shall conform to the CIM data type of the return value. Upon
 1899 successful completion, the return value of the method shall conform to the OCL expression.

1900 A string value specifying an OCL body constraint shall conform to the following formal syntax defined in
 1901 ABNF (whitespace allowed):

1902 `ocl_body_string = "body" [ocl_name] ":" ocl_statement`

1903 Where:

1904 `ocl_name` is the name of the OCL constraint.

1905 `ocl_statement` is the OCL statement of the body constraint, which defines the method return value.

1906 10.5.8 OCL derivation constraint

1907 An OCL derivation constraint is expressed as a typed OCL expression that specifies the permissible
 1908 value for a property at any time in the lifetime of the instance. The type of the expression shall conform to
 1909 the CIM data type of the property.

1910 A string value specifying an OCL derivation constraint shall conform to the following formal syntax defined
 1911 in ABNF (whitespace allowed):

1912 `ocl_derivation_string = "derive" ":" ocl_statement`

1913 Where:

1914 `ocl_statement` is the OCL statement of the derivation constraint, which defines the typed expression.

1915 10.5.9 OCL initialization constraint

1916 An OCL initialization constraint is expressed as a typed OCL expression that specifies the permissible
 1917 initial value for a property. The type of the expression shall conform to the CIM data type of the property.

1918 A string value specifying an OCL initialization constraint shall conform to the following formal syntax
 1919 defined in ABNF (whitespace allowed):

1920 `ocl_initialization_string = "init" ":" ocl_statement`

1921 Where:

1922 ocl_statement is the OCL statement of the initialization constraint, which defines the typed
1923 expression.

1924 10.6 OCL constraint examples

1925 EXAMPLE 1: For example, to check that both property x and property y cannot be NULL in any instance of a class,
1926 use the following qualifier, defined on the class:

```
1927 OCLConstraint {
1928   "inv: not (x.oclsUndefined() and y.oclsUndefined())"
1929 }
```

1930 EXAMPLE 2: The same check can be performed by first defining OCL attributes. Also, the invariant constraint is
1931 named in the following example:

```
1932 OCLConstraint {
1933   "def: xNull : Boolean = x.oclsUndefined()",
1934   "def: yNull : Boolean = y.oclsUndefined()",
1935   "inv xyNullCheck: xNull = false or yNull = false)"
1936 }
```

1937

1938 EXAMPLE:

```
1939 [OCLConstraint {
1940   "inv i1: self.p1 = self.acme_A12.r.p2"}]
1941   // Using class name ACME_A12 is required to disambiguate end name r
1942 class ACME_C1 {
1943   string p1;
1944 };
1945
1946 class ACME_C2 {
1947   [OCLConstraint {
1948     "inv i2: self.p2 = self.acme_A12.x.p1", // Using ACME_A12 is recommended
1949     "inv i3: self.p2 = self.x.p1"}] // Works, but not recommended
1950   string p2;
1951 };
1952
1953 class ACME_C3 {};
1954
1955 [Association]
1956 class ACME_A12 {
1957   ACME_C1 REF x;
1958   ACME_C2 REF r; // same name as ACME_A13::r
1959 };
1960
1961 [Association]
1962 class ACME_A13 {
1963   ACME_C1 REF y;
1964   ACME_C3 REF r; // same name as ACME_A12::r
1965 };
```

1966 EXAMPLE: The following qualifier defined on the RequestedStateChange() method of the
1967 CIM_EnabledLogicalElement class specifies that if a Job parameter is returned as not NULL, then an
1968 CIM_OwningJobElement association must exist between the CIM_EnabledLogicalElement class and the
1969 Job.

```
1970 OCLConstraint {  
1971     "post AssociatedJob: "  
1972     "not Job.oclsUndefined() "  
1973     "implies "  
1974     "self.cim_OwningJobElement.OwnedElement = Job"  
1975 }
```

1976 Derivation Example: PolicyAction has a SystemName property that must be set to the name of the
1977 system associated with CIM_PolicySetInSystem. The following qualifier defined on
1978 CIM_PolicyAction.SystemName specifies that constraint:

```
1979 OCLConstraint {  
1980     "derive: self.CIM_PolicySetInSystem.Antecedent.Name"  
1981 }
```

1982

1983
1984
1985

Annex A
(informative)
Type lattice

- 1986 The CIM Metamodel type system incorporates the types and subtypes defined in the metamodel as
1987 follows:
- 1988 • For every class C, there is an "instance of C" type, whose values shall be instances of C (including
1989 instances of any classes of C).
 - 1990 • For every structure S, there is a "value of S" type, whose values shall be values of S (including
1991 values of any subtypes of S).
 - 1992 • For every enumeration E, there is a "value of E" type, whose values shall be values of E (including
1993 values of any subtypes of E).
 - 1994 • For every class C, there is a "C Ref" type, whose values shall be references to an instance of C
1995 (including instances of any classes of C).
 - 1996 • There is a Boolean type.
 - 1997 • There is a Numeric type that is a supertype of Integer and Real.
 - 1998 • There is a Real type that is a supertype of Real32, Real64, and Real128.
 - 1999 • There is an Integer type that is a supertype of UnlimitedNatural and SignedInteger.
 - 2000 • There is an UnsignedInteger that is a supertype of UnsignedInteger
 - 2001 • There is an UnsignedInteger type that is a supertype of UInt8, UInt16, UInt32, UInt64, and UInt128.
 - 2002 • There is a SignedInteger type that is a supertype of Sint8, Sint16, Sint32, Sint64 and Sint128.
 - 2003 • There is a String type that is the supertype for OctetString.
 - 2004 • There is a Reference type.
 - 2005 • There is a DateTime type that is the supertype for TimeStamp and Duration.
 - 2006 • NOTE: A timestamp with the year field set to "0000" is interpreted as the year "1 BCE". A year field
2007 set to "0001" is interpreted as the year "1 CE".
 - 2008 • if T₁ and T₂ are non-array types, then array of T₁ is a supertype of array of T₂ if and only if T₁ is a
2009 supertype of T₂.

Annex B
(normative)
DateTime

2010
2011
2012

2013 A DateTime value is made up of multiple sub fields depending on subclass. These fields are:

- 2014 • TimeStamp::Year
- 2015 • TimeStamp::Month of Year
- 2016 • TimeStamp::Day of Month
- 2017 • TimeStamp::Hour of Day
- 2018 • TimeStamp::Minute of Hour
- 2019 • TimeStamp::Second of Minute
- 2020 • TimeStamp::Microsecond of Second
- 2021 • TimeStamp::Universal Time Correction (UTC) field
- 2022 • Duration::Days remaining
- 2023 • Duration::Hours remaining
- 2024 • Duration::Minutes remaining
- 2025 • Duration::Seconds remaining
- 2026 • Duration::Microseconds remaining

2027 The following operations are defined on DateTime sub types:

- 2028 • Arithmetic operations:
 - 2029 • Adding or subtracting a duration to or from a duration results in a duration.
 - 2030 • Adding or subtracting a duration to or from a timestamp results in a timestamp.
 - 2031 • Subtracting a timestamp from a timestamp results in a duration.
 - 2032 • Multiplying a duration by a numeric or vice versa results in a duration.
 - 2033 • Dividing a duration by a numeric results in a duration.
 - 2034 • Other arithmetic operations are not defined.
- 2035 • Comparison operations:
 - 2036 • Testing for equality of two timestamps or two durations results in a boolean value.
 - 2037 • Testing for the ordering relation (<, <=, >, >=) of two timestamps or two intervals results in a boolean value.
 - 2038
 - 2039 • Other comparison operations are not defined. For instance, comparison between a timestamp and a duration and vice versa is not defined.
 - 2040

2041 Specifications that use the definition of these operations (such as specifications for query languages)
2042 should state how undefined operations are handled.

2043 Any operations on datetime types in an expression shall be handled as if the following sequential steps
2044 were performed:

2045 20) Each datetime value is converted into a range of microsecond values, as follows:

- 2046 1) The lower bound of the range is calculated from the datetime value, with any asterisks replaced
2047 by their minimum value.
- 2048 2) The upper bound of the range is calculated from the datetime value, with any asterisks replaced
2049 by their maximum value.
- 2050 3) The basis value for timestamps is the oldest valid value (that is, 0 microseconds corresponds to
2051 00:00.000000 in the timezone with datetime offset +720, on January 1 in the year 1 BCE, using
2052 the proleptic Gregorian calendar). This definition implicitly performs timestamp normalization.
- 2053 NOTE: 1 BCE is the year before 1 CE.
- 2054 21) The expression is evaluated using the following rules for any datetime ranges:
- 2055 1) Definitions:
- 2056 • $T(x, y)$ The microsecond range for a timestamp with the lower bound x and the upper
2057 bound y
 - 2058 • $I(x, y)$ The microsecond range for a duration with the lower bound x and the upper bound y
 - 2059 • $D(x, y)$ The microsecond range for a datetime (timestamp or duration) with the lower bound x
2060 and the upper bound y
- 2061 2) Rules:
- 2062 • $I(a, b) + I(c, d) := I(a+c, b+d)$
 - 2063 • $I(a, b) - I(c, d) := I(a-d, b-c)$
 - 2064 • $T(a, b) + I(c, d) := T(a+c, b+d)$
 - 2065 • $T(a, b) - I(c, d) := T(a-d, b-c)$
 - 2066 • $T(a, b) - T(c, d) := I(a-d, b-c)$
 - 2067 • $I(a, b) * c := I(a*c, b*c)$
 - 2068 • $I(a, b) / c := I(a/c, b/c)$
 - 2069 • $D(a, b) < D(c, d) :=$ true if $b < c$, false if $a \geq d$, otherwise NULL (uncertain)
 - 2070 • $D(a, b) \leq D(c, d) :=$ true if $b \leq c$, false if $a > d$, otherwise NULL (uncertain)
 - 2071 • $D(a, b) > D(c, d) :=$ true if $a > d$, false if $b \leq c$, otherwise NULL (uncertain)
 - 2072 • $D(a, b) \geq D(c, d) :=$ true if $a \geq d$, false if $b < c$, otherwise NULL (uncertain)
 - 2073 • $D(a, b) = D(c, d) :=$ true if $a = b = c = d$, false if $b < c$ OR $a > d$, otherwise NULL (uncertain)
 - 2074 • $D(a, b) <> D(c, d) :=$ true if $b < c$ OR $a > d$, false if $a = b = c = d$, otherwise NULL (uncertain)
- 2075 These rules follow the well-known mathematical interval arithmetic. For a definition of
2076 mathematical interval arithmetic, see http://en.wikipedia.org/wiki/Interval_arithmetic.
- 2077 NOTE 1: Mathematical duration arithmetic is commutative and associative for addition and multiplication, as in
2078 ordinary arithmetic.
- 2079 NOTE 2: Mathematical duration arithmetic mandates the use of three-state logic for the result of comparison
2080 operations. A special value called "uncertain" indicates that a decision cannot be made. The special value of
2081 "uncertain" is mapped to the NULL value in datetime comparison operations.
- 2082 22) Overflow and underflow condition checking is performed on the result of the expression, as follows:
- 2083 1) For timestamp results:
- 2084 • A timestamp older than the oldest valid value in the timezone of the result produces an
2085 arithmetic underflow condition.

- 2086 • A timestamp newer than the newest valid value in the timezone of the result produces an arithmetic overflow condition.
- 2087
- 2088 2) For interval results:
- 2089 • A negative interval produces an arithmetic underflow condition.
- 2090 • A positive interval greater than the largest valid value produces an arithmetic overflow condition.
- 2091 Specifications using these operations (for instance, query languages) should define how these conditions are handled.
- 2092
- 2093 23) If the result of the expression is a datetime type, the microsecond range is converted into a valid datetime value such that the set of asterisks (if any) determines a range that matches the actual result range or encloses it as closely as possible. The GMT timezone shall be used for any timestamp results.
- 2094
- 2095
- 2096
- 2097 NOTE: For most fields, asterisks can be used only with the granularity of the entire field.

2098 **Table 10: Evaluation of DateTime Expressions**

Expression	Result
"20051003110000.000000+000"+"00000000002233.000000:000"	"20051003112233.000000+000"
"20051003110000.*****+000"+"00000000002233.000000:000"	"20051003112233.*****+000"
"20051003110000.*****+000"+"00000000002233.000000:000"	"200510031122** *****+000"
"20051003110000.*****+000"+"00000000002233.*****:000"	"200510031122** *****+000"
"20051003110000.*****+000"+"00000000005959.*****:000"	"20051003*****.*****+000"
"20051003110000.*****+000"+"000000000022** *****:000"	"2005100311****.*****+000"
"20051003112233.000000+000"-"00000000002233.000000:000"	"20051003110000.000000+000"
"20051003112233.*****+000"-"00000000002233.000000:000"	"20051003110000.*****+000"
"20051003112233.*****+000"-"00000000002233.000000*:000"	"20051003110000.*****+000"
"20051003112233.*****+000"-"00000000002232.*****:000"	"200510031100** *****+000"
"20051003112233.*****+000"-"00000000002233.*****:000"	"20051003*****.*****+000"
"20051003060000.000000-300"+"00000000002233.000000:000"	"20051003112233.000000+000"
"20051003060000.*****-300"+"00000000002233.000000:000"	"20051003112233.*****+000"
"000000000011** *****:000"*60	"0000000011****.*****:000"
60timesaddingup "000000000011** *****:000"	"0000000011****.*****:000"
"20051003112233.000000+000"="20051003112233.000000+000"	true
"20051003122233.000000+060"="20051003112233.000000+000"	true
"20051003112233.*****+000"="20051003112233.*****+000"	true
"20051003112233.*****+000"="200510031122** *****+000"	NULL(uncertain)
"20051003112233.*****+000"="20051003112234.*****+000"	false
"20051003112233.*****+000"<"20051003112234.*****+000"	true
"20051003112233.5*****+000"<"20051003112233.*****+000"	NULL(uncertain)

- 2099 A datetime value is valid if the value of each single field is in the valid range. Valid values shall not be rejected by any validity checking within the CIM infraclass.
- 2100
- 2101 Within these valid ranges, some values are defined as reserved. Values from these reserved ranges shall not be interpreted as points in time or durations.
- 2102

2103

2104

2105

2106

2107

2108

2109

2110

2111

2112

2113

2114

2115

2116

Annex C
(normative)
Backwards compatibility rules for schema modifications

Table 11 lists modifications to the CIM schemas in final status that cause a major version number change. Preliminary models are allowed to evolve based on implementation experience. These modifications change application behavior and/or customer code. Therefore, they force a major version update and are discouraged. Table 11 is an exhaustive list of the possible modifications based on current CIM experience and knowledge. Items could be added as new issues are raised and CIM standards evolve.

Alterations beyond those listed in Table 11 are considered interface-preserving and require the minor version number to be incremented. Updates/errata are not classified as major or minor in their impact, but they are required to correct errors or to coordinate across standards bodies.

Table 11: Changes that Increment the CIM Schema Major Version

Description	Explanation or Exceptions
Deletion of a NamedElement	
Data type change to a Property or Parameter	
Signature change to a Method	
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, class, or enumeration upwards in the inheritance hierarchy; that is, the removal of general element from the inheritance hierarchy	The removal of general elements deletes properties or methods. New classes, classes, or enumeration can be inserted as generalizations as a minor change or update. Inserted classes shall not change keys or add required properties. Inserted class shall not add required properties
Changing a concrete class to Abstract	
Change of an association reference 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 qualifier	
For input Parameters or writeable Properties, restricting the allowable range of values, (including disallowance of no value if it had been allowed.)	Changing to disallow values from previously allowed value ranges to be passed to an input parameter or to be written to a property may break existing clients that pass those values under the prior definition. The addition of allowed values for method input parameters and properties that may only be written is a compatible change, as clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the defining element. 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.

Description	Explanation or Exceptions
For output Parameters (including return) or readable Properties, increasing the allowable range of values, (including allowance of no value if it had been disallowed.)	<p>Changing to allow values from previously disallowed value ranges to be returned by an output parameter, a method return value, or a property that may be read may break existing clients that relied on the prior guarantee.</p> <p>A restriction to the range of allowed values to method output parameters, method return values and properties that may only be read is considered a compatible change, as clients written to the new behavior are expected to determine whether they communicate with the old or new behavior of the server.</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.</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	
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	

2118
2119
2120

Annex D (normative) UCS and Unicode

2121 [ISO/IEC 10646:2003](#) defines the Universal Multiple-Octet Coded Character Set (UCS). [The Unicode](#)
2122 [Standard](#) defines Unicode. This clause gives a short overview on UCS and Unicode for the scope of this
2123 document, and defines which of these standards is used by this document.

2124 Even though these two standards define slightly different terminology, they are consistent in the
2125 overlapping area of their scopes. Particularly, there are matching releases of these two standards that
2126 define the same UCS/Unicode character repertoire. In addition, each of these standards covers some
2127 scope that the other does not.

2128 This document uses [ISO/IEC 10646:2003](#) and its terminology. [ISO/IEC 10646:2003](#) references some
2129 annexes of [The Unicode Standard](#). Where it improves the understanding, this document also states terms
2130 defined in [The Unicode Standard](#) in parenthesis.

2131 Both standards define two layers of mapping:

- 2132 • Characters (Unicode Standard: abstract characters) are assigned to UCS code positions (Unicode
2133 Standard: code points) in the value space of the integers 0 to 0x10FFFF.

2134 In this document, these code positions are referenced using the U+xxxxxx format defined in [ISO/IEC](#)
2135 [10646:2003](#). In that format, the aforementioned value space would be stated as U+0000 to
2136 U+10FFFF.

2137 Not all UCS code positions are assigned to characters; some code positions have a special purpose
2138 and most code positions are available for future assignment by the standard.

2139 For some characters, there are multiple ways to represent them at the level of code positions. For
2140 example, the character "LATIN SMALL LETTER A WITH GRAVE" (à) can be represented as a
2141 single *precomposed character* at code position U+00E0 (à), or as a sequence of two characters: A
2142 *base character* at code position U+0061 (a), followed by a *combination character* at code position
2143 U+0300 (ˆ). [ISO/IEC 10646:2003](#) references [The Unicode Standard, Version 5.2.0, Annex #15:](#)
2144 [Unicode Normalization Forms](#) for the definition of *normalization forms*. That annex defines four
2145 normalization forms, each of which reduces such multiple ways for representing characters in the
2146 UCS code position space to a single and thus predictable way. The [Character Model for the World](#)
2147 [Wide Web 1.0: Normalization](#) recommends using *Normalization Form C* (NFC) defined in that annex
2148 for all content, because this form avoids potential interoperability problems arising from the use of
2149 canonically equivalent, yet differently represented, character sequences in document formats on the
2150 Web. NFC uses precomposed characters where possible, but not all characters of the UCS
2151 character repertoire can be represented as precomposed characters.

- 2152 • UCS code position values are assigned to binary data values of a certain size that can be stored in
2153 computer memory.

2154 The set of rules governing the assignment of a set of UCS code points to a set of to binary data
2155 values is called a *coded representation form* (Unicode Standard: *encoding form*). Examples are
2156 UCS-2, UTF-16 or UTF-8.

2157 Two sequences of binary data values representing UCS characters that use the same normalization form
2158 and the same coded representation form can be compared for equality of the characters by performing a
2159 binary (e.g., octet-wise) comparison for equality.

2160
2161
2162

Annex E (normative) Comparison of values

- 2163 This annex defines comparison of values for equality and ordering.
- 2164 Values of boolean datatypes shall be compared for equality and ordering as if "true" was 1 and "false"
2165 was 0 and the mathematical comparison rules for integer numbers were used on those values.
- 2166 Comparison is supported between all numeric types. When comparisons are made between different
2167 numeric types, comparison is performed using the type with the greater precision.
- 2168 Values of integer number datatypes shall be compared for equality and ordering according to the
2169 mathematical comparison rules for the integer numbers they represent.
- 2170 Values of real number datatypes shall be compared for equality and ordering according to the rules
2171 defined in [ANSI/IEEE 754-1985](#).
- 2172 Values of the string datatypes shall be compared for equality on a UCS character basis, by using the
2173 string identity matching rules defined in chapter 4 "String Identity Matching" of the [Character Model for the
2174 World Wide Web 1.0: Normalization](#) specification.
- 2175 In order to minimize the processing involved in UCS normalization, string typed values should be stored
2176 and transmitted in Normalization Form C (NFC) as defined in [The Unicode Standard, Version 5.2.0](#).
2177 [Annex #15: Unicode Normalization Forms](#). This allows skipping the costly normalization when comparing
2178 the strings.
- 2179 This document does not define an order between values of the string datatypes, since UCS ordering rules
2180 may be compute intensive and their usage should be decided on a case by case basis. The ordering of
2181 the "Common Template Table" defined in [ISO/IEC 14651:2007](#) provides a reasonable default ordering of
2182 UCS strings for human consumption. However, an ordering based on the UCS code positions, or even
2183 based on the octets of a particular UCS coded representation form is typically less compute intensive and
2184 may be sufficient, for example when no human consumption of the ordering result is needed.
- 2185 Two values of the octetstring datatype shall be considered equal if they contain the same number of
2186 octets and have equal octets in each octet pair in the sequences. An octet sequence S1 shall be
2187 considered less than an octet sequence S2, if the first pair of different octets, reading from left to right, is
2188 beyond the end of S1 or has an octet in S1 that is less than the octet in S2. This comparison rule yields
2189 the same results as the comparison rule defined for the strcmp() function in [IEEE Std 1003.1, 2004
2190 Edition](#).
- 2191 Two values of the reference datatype shall be considered equal if they resolve to the same class instance
2192 in the same NamingContext. This document does not define an order between two values of the
2193 reference datatype.
- 2194 Two values of the datetime datatype shall be compared based on the time duration or point in time they
2195 represent, according to mathematical comparison rules for these numbers. As a result, two datetime
2196 values that represent the same point in time using different timezone offsets are considered equal.
- 2197 Two values of compatible datatypes that both have no value, (i.e. are NULL), shall be considered equal.
2198 This document does not define an order between two values of compatible datatypes where one has a
2199 value, and the other does not.
- 2200 Two array values of compatible datatypes shall be considered equal if they contain the same number of
2201 array entries and in each pair of array entries, the two array entries are equal. This document does not
2202 define an order between two array values.

2203

2204

Annex F
(normative)
Programmatic units

2205
2206
2207

2208 This annex defines the concept and syntax of a programmatic unit, which is an expression of a unit of
2209 measure for programmatic access. It makes it easy to recognize the base units of which the actual unit is
2210 made, as well as any numerical multipliers. Programmatic units are used as a value for the PUnit qualifier
2211 and also as a value for any (string typed) CIM elements that represent units. The boolean IsPUnit qualifier
2212 is used to declare that a string typed element follows the syntax for programmatic units.

2213 Programmatic units must be processed case-sensitively and white-space-sensitively.

2214 As defined in the Augmented BNF (ABNF) syntax, the programmatic unit consists of a base unit that is
2215 optionally followed by other base units that are each either multiplied or divided into the first base unit.
2216 Furthermore, two optional multipliers can be applied. The first is simply a scalar, and the second is an
2217 exponential number consisting of a base and an exponent. The optional multipliers enable the
2218 specification of common derived units of measure in terms of the allowed base units. The base units
2219 defined in this clause include a superset of the SI base units. When a unit is the empty string, the value
2220 has no unit; that is, it is dimensionless. The multipliers must be understood as part of the definition of the
2221 derived unit; that is, scale prefixes of units are replaced with their numerical value. For example,
2222 "kilometer" is represented as "meter * 1000", replacing the "kilo" scale prefix with the numerical factor
2223 1000.

2224 A string representing a programmatic unit must follow the format defined by the programmatic-unit ABNF
2225 rule in the syntax defined in this annex. This format supports any type of unit, including SI units, United
2226 States units, and any other standard or non-standard units.

2227 The ABNF syntax is defined as follows. This ABNF explicitly states any whitespace characters that may
2228 be used, and whitespace characters in addition to those are not allowed.

```

2229 programmatic-unit = ( "" / base-unit *( [WS] multiplied-base-unit
2230 *( [WS] divided-base-unit ) [ [WS] modifier1 ] [ [WS] modifier2 ] )
2231
2232 multiplied-base-unit = "*" [WS] base-unit
2233
2234 divided-base-unit = "/" [WS] base-unit
2235
2236 modifier1 = operator [WS] number
2237
2238 modifier2 = operator [WS] base [WS] "^" [WS] exponent
2239
2240 operator = "*" / "/"
2241
2242 number = ["+" / "-"] positive-number
2243
2244 base = positive-whole-number
2245
2246 exponent = ["+" / "-"] positive-whole-number
2247
2248 positive-whole-number = NON-ZERO-DIGIT *( DIGIT )
2249

```

2250 positive-number = positive-whole-number
 2251 / ((positive-whole-number / ZERO) "." *(DIGIT))
 2252
 2253 base-unit = simple-name / decibel-base-unit
 2254
 2255 simple-name = FIRST-UNIT-CHAR *([S] UNIT-CHAR)
 2256
 2257 decibel-base-unit = "decibel" [[S] "(" [S] simple-name [S] ")"]
 2258
 2259 FIRST-UNIT-CHAR = UPPERALPHA / LOWERALPHA / UNDERSCORE / UCS0080TOFFEF
 2260 ; DEPRECATED: The use of the UCS0080TOFFEF ABNF rule within
 2261 ; the FIRST-UNIT-CHAR ABNF rule is deprecated since
 2262 ; version 2.6.0 of this document.
 2263
 2264 UNIT-CHAR = FIRST-UNIT-CHAR / S / HYPHEN / DIGIT
 2265
 2266 ZERO = "0"
 2267
 2268 NON-ZERO-DIGIT = ("1"..."9")
 2269
 2270 DIGIT = ZERO / NON-ZERO-DIGIT
 2271
 2272 WS = (S / TAB / NL)
 2273
 2274 S = U+0020 ; " " (space)
 2275
 2276 TAB = U+0009 ; "\t" (tab)
 2277
 2278 NL = U+000A ; "\n" (newline, linefeed)
 2279
 2280 HYPHEN = U+000A ; "-" (hyphen, minus)
 2281
 2282 UPPERALPHA = U+0041...U+005A ; "A" ... "Z"
 2283
 2284 LOWERALPHA = U+0061...U+007A ; "a" ... "z"
 2285
 2286 UNDERSCORE = U+005F ; "_"
 2287
 2288 UCS0080TOFFEF is any assigned UCS character with code positions
 2289 in the range U+0080..U+FFEF
 2290

2291 For example, a speedometer may be modeled so that the unit of measure is kilometers per hour. It is
 2292 necessary to express the derived unit of measure "kilometers per hour" in terms of the allowed base units
 2293 "meter" and "second". One kilometer per hour is equivalent to

2294 1000 meters per 3600 seconds

2295 or

2296 one meter / second / 3.6

2297 so the programmatic unit for "kilometers per hour" is expressed as: "meter / second / 3.6", using the
 2298 syntax defined here.

2299 Other examples are as follows:

- 2300 "meter * meter * 10^-6" → square millimeters
- 2301 "byte * 2^10" → kBytes as used for memory ("kibobyte")
- 2302 "byte * 10^3" → kBytes as used for storage ("kilobyte")
- 2303 "dataword * 4" → QuadWords
- 2304 "decibel(m) * -1" → -dBm
- 2305 "second * 250 * 10^-9" → 250 nanoseconds
- 2306 "foot * foot * foot / minute" → cubic feet per minute, CFM
- 2307 "revolution / minute" → revolutions per minute, RPM
- 2308 "pound / inch / inch" → pounds per square inch, PSI
- 2309 "foot * pound" → foot-pounds

2310 In the "PU Base Unit" column, Table 12 defines the allowed values for the base-unit ABNF rule in the
 2311 syntax, as well as the empty string indicating no unit. The "Symbol" column recommends a symbol to be
 2312 used in a human interface. The "Calculation" column relates units to other units. The "Quantity" column
 2313 lists the physical quantity measured by the unit.

2314 The base units in Table 12 consist of the SI base units and the SI derived units amended by other
 2315 commonly used units. "SI" is the international abbreviation for the International System of Units (French:
 2316 "Système International d'Unites"), defined in ISO 1000:1992. Also, ISO 1000:1992 defines the notational
 2317 conventions for units, which are used in Table 12.

2318 **Table 12: 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 m²/m²	Solid angle
bit	bit		Quantity of information

PU Base Unit	Symbol	Calculation	Quantity
byte	B	1 B = 8 bit	Quantity of information
dataword	word	1 word = N bit	Quantity of information. The number of bits depends on the computer architecture.
meter	m	SI base unit	Length (The corresponding ISO SI unit is "metre.")
inch	in	1 in = 0.0254 m	Length
rack unit	U	1 U = 1.75 in	Length (height unit used for computer components, as defined in EIA-310)
foot	ft	1 ft = 12 in	Length
yard	yd	1 yd = 3 ft	Length
mile	mi	1 mi = 1760 yd	Length (U.S. land mile)
liter	l	1000 l = 1 m ³	Volume (The corresponding ISO SI unit is "litre.")
fluid ounce	fl.oz	33.8140227 fl.oz = 1 l	Volume for liquids (U.S. fluid ounce)
liquid gallon	gal	1 gal = 128 fl.oz	Volume for liquids (U.S. liquid gallon)
mole	mol	SI base unit	Amount of substance
kilogram	kg	SI base unit	Mass
ounce	oz	35.27396195 oz = 1 kg	Mass (U.S. ounce, avoirdupois ounce)
pound	lb	1 lb = 16 oz	Mass (U.S. pound, avoirdupois pound)
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

PU Base Unit	Symbol	Calculation	Quantity
degree celsius	°C	1 °C = 1 K (diff)	Thermodynamic temperature
degree fahrenheit	°F	1 °F = 5/9 K (diff)	Thermodynamic temperature
kelvin	K	SI base unit	Thermodynamic temperature, color temperature
candela	cd	SI base unit	Luminous intensity
lumen	lm	1 lm = 1 cd·sr	Luminous flux
nit	nit	1 nit = 1 cd/m ²	Luminance
lux	lx	1 lx = 1 lm/m ²	Illuminance
newton	N	1 N = 1 kg·m/s ²	Force
pascal	Pa	1 Pa = 1 N/m ²	Pressure
bar	bar	1 bar = 100000 Pa	Pressure
decibel(A)	dB(A)	1 dB(A) = 20 lg (p/p ₀)	Loudness of sound, relative to reference sound pressure level of p ₀ = 20 μPa in gases, using frequency weight curve (A)
decibel(C)	dB(C)	1 dB(C) = 20 · lg (p/p ₀)	Loudness of sound, relative to reference sound pressure level of p ₀ = 20 μPa in gases, using frequency weight curve (C)
joule	J	1 J = 1 N·m	Energy, work, torque, quantity of heat
watt	W	1 W = 1 J/s = 1 V · A	Power, radiant flux. In electric power technology, the real power (also known as active power or effective power or true power)
volt ampere	VA	1 VA = 1 V · A	In electric power technology, the apparent power
volt ampere reactive	var	1 var = 1 V · A	In electric power technology, the reactive power (also known as imaginary power)
decibel(m)	dBm	1 dBm = 10 · lg (P/P ₀)	Power, relative to reference power of P ₀ = 1 mW
british thermal unit	BTU	1 BTU = 1055.056 J	Energy, quantity of heat. The ISO definition of BTU is used here, out of multiple definitions.
ampere	A	SI base unit	Electric current, magnetomotive force
coulomb	C	1 C = 1 A·s	Electric charge

PU Base Unit	Symbol	Calculation	Quantity
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

2319
2320
2321

Annex G (normative) MappingStrings formats

2322 G.1 Mapping entities of other information models to CIM

2323 The MappingStrings qualifier can be used to map entities of other information models to CIM or to
2324 express that a CIM element represents an entity of another information model. Several mapping string
2325 formats are defined in this clause to use as values for this qualifier. The CIM schema shall use only the
2326 mapping string formats defined in this document. Extension schemas should use only the mapping string
2327 formats defined in this document.

2328 The mapping string formats defined in this document conform to the following formal syntax defined in
2329 ABNF:

```
2330 mappingstrings_format = mib_format / oid_format / general_format / mif_format
```

2331 NOTE: As defined in the respective clauses, the "MIB", "OID", and "MIF" formats support a limited form of extensibility
2332 by allowing an open set of defining bodies. However, the syntax defined for these formats does not allow variations
2333 by defining body; they need to conform. A larger degree of extensibility is supported in the general format, where the
2334 defining bodies may define a part of the syntax used in the mapping.

2335 G.2 SNMP-related mapping string formats

2336 The two SNMP-related mapping string formats, Management Information Base (MIB) and globally unique
2337 object identifier (OID), can express that a CIM element represents a MIB variable. As defined in
2338 [RFC1155](#), a MIB variable has an associated variable name that is unique within a MIB and an OID that is
2339 unique within a management protocol.

2340 The "MIB" mapping string format identifies a MIB variable using naming authority, MIB name, and variable
2341 name. It may be used only on CIM properties, parameters, or methods. The format is defined as follows,
2342 using ABNF:

```
2343 mib_format = "MIB" "." mib_naming_authority "|" mib_name "." mib_variable_name
```

2344 Where:

```
2345 mib_naming_authority = 1*(stringChar)
```

2346 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical
2347 bar (|) characters are not allowed.

```
2348 mib_name = 1*(stringChar)
```

2349 is the name of the MIB as defined by the MIB naming authority (for example, "HOST-RESOURCES-
2350 MIB"). The dot (.) and vertical bar (|) characters are not allowed.

```
2351 mib_variable_name = 1*(stringChar)
```

2352 is the name of the MIB variable as defined in the MIB (for example, "hrSystemDate"). The dot (.)
2353 and vertical bar (|) characters are not allowed.

2354 The MIB name should be the ASN.1 module name of the MIB (that is, not the RFC number). For example,
2355 instead of using "RFC1493", the string "BRIDGE-MIB" should be used.

2356 EXAMPLE:

```
2357 [MappingStrings { "MIB.IETF|HOST-RESOURCES-MIB.hrSystemDate" }]
```

```
2358 datetime LocalDateTime;
```

2359 The "OID" mapping string format identifies a MIB variable using a management protocol and an object
 2360 identifier (OID) within the context of that protocol. This format is especially important for mapping
 2361 variables defined in private MIBs. It may be used only on CIM properties, parameters, or methods. The
 2362 format is defined as follows, using ABNF:

```
2363 oid_format = "OID" "." oid_naming_authority "|" oid_protocol_name "." oid
```

2364 Where:

```
2365 oid_naming_authority = 1*(stringChar)
```

2366 is the name of the naming authority defining the MIB (for example, "IETF"). The dot (.) and vertical
 2367 bar (|) characters are not allowed.

```
2368 oid_protocol_name = 1*(stringChar)
```

2369 is the name of the protocol providing the context for the OID of the MIB variable (for example,
 2370 "SNMP"). The dot (.) and vertical bar (|) characters are not allowed.

```
2371 oid = 1*(stringChar)
```

2372 is the object identifier (OID) of the MIB variable in the context of the protocol (for example,
 2373 "1.3.6.1.2.1.25.1.2").

2374 EXAMPLE:

```
2375 [MappingStrings { "OID.IETF|SNMP.1.3.6.1.2.1.25.1.2" }]
```

```
2376 datetime LocalDateTime;
```

2377 For both mapping string formats, the name of the naming authority defining the MIB shall be one of the
 2378 following:

- 2379 • The name of a standards body (for example, IETF), for standard MIBs defined by that standards
 2380 body
- 2381 • A company name (for example, Acme), for private MIBs defined by that company

2382 G.3 General mapping string format

2383 This clause defines the mapping string format, which provides a basis for future mapping string formats.
 2384 Future mapping string formats defined in this document should be based on the general mapping string
 2385 format. A mapping string format based on this format shall define the kinds of CIM elements with which it
 2386 is to be used.

2387 The format is defined as follows, using ABNF. The division between the name of the format and the
 2388 actual mapping is slightly different than for the "MIF", "MIB", and "OID" formats:

```
2389 general_format = general_format_fullname "|" general_format_mapping
```

2390 Where:

```
2391 general_format_fullname = general_format_name "." general_format_defining_body
```

```
2392 general_format_name = 1*(stringChar)
```

2393 is the name of the format, unique within the defining body. The dot (.) and vertical bar (|)
 2394 characters are not allowed.

```
2395 general_format_defining_body = 1*(stringChar)
```

2396 is the name of the defining body. The dot (.) and vertical bar (|) characters are not allowed.

2397 `general_format_mapping = 1*(stringChar)`

2398 is the mapping of the qualified CIM element, using the named format.

2399 The text in Table 13 is an example that defines a mapping string format based on the general mapping
2400 string format.

2401 **Table 13: Example MappingStrings mapping**

General Mapping String Formats Defined for InfiniBand Trade Association (IBTA)
<p>IBTA defines the following mapping string formats, which are based on the general mapping string format:</p> <p><code>"MAD.IBTA"</code></p> <p>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:</p> <p><code>general_format_fullname = "MAD" "." "IBTA"</code></p> <p><code>general_format_mapping = mad_class_name " " mad_attribute_name</code></p> <p>Where:</p> <p><code>mad_class_name = 1*(stringChar)</code> is the name of the MAD class. The dot (.) and vertical bar () characters are not allowed.</p> <p><code>mad_attribute_name = 1*(stringChar)</code> is the name of the MAD attribute, which is unique within the MAD class. The dot (.) and vertical bar () characters are not allowed.</p>

2402

2403
2404
2405

Annex H
(informative)
Modeling guidelines

2406 The following are general guidelines for CIM modeling:

- 2407 • Method descriptions are recommended and should, at a minimum, indicate the method's side
2408 effects (pre- and post-conditions).
- 2409 • Leading underscores in identifiers are to be discouraged and not used at all in the standard
2410 schemas.
- 2411 • It is generally recommended that class names not be reused as part of property or method
2412 names. Property and method names are already unique within their defining class.
- 2413 • To enable information sharing among different CIM implementations, the MaxLen qualifier
2414 should be used to specify the maximum length of string properties.
- 2415 • When extending a schema (i.e., CIM schema or extension schema) with new classes, existing
2416 classes should be considered as superclasses of such new classes as appropriate, in order to
2417 increase schema consistency.

2418 **H.1 SQL reserved words**

2419 Avoid using SQL reserved words in class and property names. This restriction particularly applies to
 2420 property names because class names are generally prefixed by the schema name, making a clash with a
 2421 reserved word unlikely. The current set of SQL reserved words is as follows:

2422 From sql1992.txt:

AFTER	ALIAS	ASYN	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	SQL EXCEPTION	SQLWARNING	STRUCTURE
TEST	THERE	TRIGGER	TYPE
UNDER	VARIABLE	VIRTUAL	VISIBLE
WAIT	WHILE	WITHOUT	

2423 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	HOURL	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			

2424 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		

2425 From Annex E of sql3part4.txt:

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

2426

2427

(j | j=value.at(i))

2428
2429
2430

**Annex I
(informative)
Change log**

Version	Date	Description
1	1997-04-09	First Public Release, CIM Infraclass specification
2.2	1999-06-14	Released as Final Standard
2.2.1000	2003-06-07	Released as Final Standard
2.3	2005-10-04	Released as Final Standard
2.4	-----	Was not released
2.5	2005-05-01	Released as a DMTF Standard
2.6	2010-03-17	Released as a DMTF Standard
		•
3.0.0		Name changed to CIM Metamodel specification

2431

Bibliography

- 2432 DMTF DSP0200, CIM operations over HTTP, Version 1.3
2433 http://www.dmtf.org/standards/published_documents/DSP0200_1.3.pdf
- 2434 DMTF DSP0201, Specification for the Representation of CIM in XML, Version 2.3
2435 http://www.dmtf.org/standards/published_documents/DSP0201_2.3.pdf
- 2436 ISO/IEC 19757-2:2008, Information technology -- Document Schema Definition Language (DSDL) -- Part
2437 2: Regular-grammar-based validation -- RELAX NG,
2438 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52348
- 2439 IETF, RFC2068, Hypertext Transfer Protocol – HTTP/1.1, <http://tools.ietf.org/html/rfc2068>
- 2440 IETF, RFC1155, Structure and Identification of Management Information for TCP/IP-based Internets,
2441 <http://tools.ietf.org/html/rfc1155>
- 2442 IETF, RFC2253, Lightweight Directory Access Protocol (v3): UTF-8 String Representation Of
2443 Distinguished Names, <http://tools.ietf.org/html/rfc2253>
- 2444 The Unicode Consortium: The Unicode Standard, <http://www.unicode.org>
- 2445 W3C, Character Model for the World Wide Web 1.0: Normalization, Working Draft, 27 October 2005,
2446 <http://www.w3.org/TR/charmod-norm/>
- 2447 W3C, XML Schema Part 0: Primer Second Edition, W3C Recommendation, 28 October 2004,
2448 <http://www.w3.org/TR/xmlschema-0/>
- 2449 DMTF Architecture WG. (2011). DSP2026 Distributed Management Infrastructure Overview. Version 1.0,
2450 http://www.dmtf.org/sites/default/files/standards/documents/DSP202x_1.0.0.pdf
- 2451
- 2452