INTERNATIONAL STANDARD

**ISO 13606-2**

Second edition
2019-06

# Health informatics — Electronic health record communication —

Part 2:
**Archetype interchange specification**

*Informatique de santé — Communication du dossier de santé informatisé —*

*Partie 2: Spécification d'échange d'archétype*

Reference number
ISO 13606-2:2019(E)

## Contents

# Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/TC 215, *Health Informatics*.

This second edition cancels and replaces the first edition (ISO 13606-2:2008), which has been technically revised. The main changes compared to the previous edition are as follows:

— Introduction of new internal coding scheme, consisting of id-codes, at-codes and ac-codes.

— Replace string archetype identifier with multi-part, namespace identifier.

— Addition of explicit value-sets replacing in-line value sets in the terms and definitions.

— Renaming archetype ontology section to terminology.

— Expression of all external term bindings as URIs following IHTSDO format.

— Introduction of 'tuple' constraints for co-varying attributes within Quantity, Ordinal structures.

— Re-engineering of all primitive constrainer types, i.e. C_STRING, C_DATE etc.

— Removal of the Archetype Profile specification.

— Full specialisation support: the addition of an attribute to the C_ATTRIBUTE class, allowing the inclusion of a path that enables specialised archetype redefinitions deep within a structure.

— Addition of node-level annotations.

— Structural simplification of archetype ontology section.

— The name of the invariant section has been changed to rules, to better reflect its purpose.

— A template is now just an archetype.

A list of all parts in the ISO 13606 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

# Introduction

This document is part of a five-part standard series, published jointly by CEN and ISO through the Vienna Agreement. In this document dependency upon any of the other parts of this series is explicitly stated where it applies.

Comprehensive, multi-enterprise and longitudinal electronic health records will often in practice be achieved through the joining up of multiple clinical applications, databases (and increasingly devices) that are each tailored to the needs of individual conditions, specialties or enterprises.

This requires that Electronic Health Record (EHR) data from diverse systems be capable of being mapped to and from a single comprehensive representation, which is used to underpin interfaces and messages within a distributed network (federation) of EHR systems and services. This common representation has to be sufficiently generic and rich to represent any conceivable health record data, comprising part or all of an EHR (or a set of EHRs) being communicated.

The approach adopted in the ISO 13606 standards series, underpinned by international research on the EHR, has been to define a rigorous and generic Reference Model that is suitable for all kinds of data and data structures within an EHR, and in which all labelling and context information is an integral part of each construct. An EHR Extract (as defined in ISO 13606-1) will contain all of the names, structure and context required for it to be interpreted faithfully on receipt even if its organisation and the nature of the clinical content have not been "agreed" in advance.

However, the wide-scale sharing of health records, and their meaningful analysis across distributed sites, also requires that a consistent approach is used for the clinical (semantic) data structures that will be communicated via the Reference Model, so that equivalent clinical information is represented consistently. This is necessary in order for clinical applications and analysis tools safely to process EHR data that have come from heterogeneous sources.

## 0.1 Archetypes

The challenge for EHR interoperability is therefore to devise a generalised approach to representing every conceivable kind of health record data structure in a consistent way. This needs to cater for records arising from any profession, speciality or service, whilst recognising that the clinical data sets, value sets, templates etc. required by different health care domains will be diverse, complex and will change frequently as clinical practice and medical knowledge advance. This requirement is part of the widely acknowledged health informatics challenge of semantic interoperability.

The approach adopted by this standard series distinguishes a Reference Model, used to represent the generic properties of health record information, and Archetypes (conforming to an Archetype Model), which are meta-data used to define patterns for the specific characteristics of the clinical data that represent the requirements of each particular profession, speciality or service.

**The Reference Model** is specified as an Open Distributed Processing (ODP) Information Viewpoint model, representing the global characteristics of health record components, how they are aggregated, and the context information required to meet ethical, legal and provenance requirements. In the 13606 standards series, the Reference Model is defined in Part 1. This model defines the set of classes that form the generic building blocks of the EHR. It reflects the stable characteristics of an electronic health record, and would be embedded in a distributed (federated) EHR environment as specific messages or interfaces (as specified in Part 5 of this standard series).

**Archetypes** are effectively pre-coordinated combinations of named RECORD_COMPONENT hierarchies that are agreed within a community in order to ensure semantic interoperability, data consistency and data quality.

For an EHR_EXTRACT, as defined in ISO 13606-1, an archetype specifies (and effectively constrains) a particular hierarchy of RECORD_COMPONENT sub-classes, defining or constraining their names and other relevant attribute values, optionality and multiplicity at any point in the hierarchy, the datatypes and value ranges that ELEMENT data values can take, and might include other dependency constraints. Archetype instances themselves conform to a formal model, known as an Archetype Model (which is a constraint model, also specified as an ODP Information Viewpoint Model). Although the Archetype Model is stable, individual archetype instances can be revised or succeeded by others as clinical practice evolves. Version control ensures that new revisions do not invalidate data created with previous revisions.

Archetypes can be used within EHR systems to govern the EHR data committed to a repository. However, for the purposes of this interoperability standard series, no assumption is made about the use of archetypes within the EHR Provider system whenever this standard series is used for EHR communication. It is assumed that the original EHR data, if not already archetyped, can be mapped to a set of archetypes, if desired, when generating the EHR_EXTRACT.

The reference model defined in ISO 13606-1 has a property that can be used to specify the archetype to which any RECORD_COMPONENT within an EHR_EXTRACT conforms. The class RECORD_COMPONENT includes an attribute *archetype_id* to identify the archetype and node to which that RECORD_COMPONENT conforms.

Part 3 of this standard series includes a set of Reference Archetypes: which are base archetypes that are likely to be specialised further before they are used. Those archetypes are example instances of this Archetype Model.

The Archetype Model specified in this document was originally developed by the openEHR Foundation, which publishes its archetypes using Archetype Definition Language, conforming to this Archetype Model, referenced within Annex A. The Archetype Model has been the subject of collaborative updating to incorporate the requirements and modelling inputs from the Clinical Information Modeling Initiative (CIMI). CIMI is in the process of submitting a modelling language (Archetype Modeling Language, AML) to the Object Management Group. AML also aligns to this Archetype Model.

## 0.2 Archetype datatypes

It should be noted that ISO 13606-1 and ISO 13606-2 use datatypes for different purposes.

Part 1 defines datatypes to represent the properties of the Reference Model, as a profile of ISO 21090, in 5.3. It separately defines in Clause 7 the data types that can be the values of Element, also a subset of ISO 21090. All these datatypes are finally expressed in terms of the so-called “primitive” datatypes (Integer, Real, String, Boolean, Date/Time/Datetime).

Part 2 uses the same set of primitive datatypes to represent the properties of the Archetype Object Model. Additionally, Part 2 defines a set of classes that allow defining constraints over primitive datatypes of Part 1. These constraining classes are shown in Figure 9 of Part 2, as descendants of the C_PRIMITIVE_OBJECT class.

A single Part 1 complex datatype (e.g. PHYSICAL_QUANTITY) can be constrained by a combination of the constraining classes of the Archetype Object Model, defining constraints on both the complex and primitive datatypes it contains. Thus, Part 1 complex datatypes are treated as classes when defining constraints with Part 2, while Part 1 primitive data types are constrained by the C_PRIMITIVE_OBJECT hierarchy.

An example of a PHYSICAL_QUANTITY archetype can be seen in the example below. In this example, the value on a PHYSICAL_QUANTITY shall be between 0.0 and 1000.0 and their units shall be UCUM ‘mm[Hg]’ code.

```
PHYSICAL_QUANTITY matches {
        value matches {|0.0..<1000.0|}
        units matches {
            CODED_SIMPLE matches {
                        value matches {"mm[Hg]"}
            }
        }
    }
```

This example archetype, expressed in terms of the Archetype Object Model, would have the structure shown in Table 1.

**Table 1 — Example structure for representing physical quantity**

| **Reference Model class, attribute or primitive value** | **Archetype Model constraining class** |
|---|---|
| PHYSICAL_QUANTITY | C_COMPLEX_OBJECT |
|     value |     C_ATTRIBUTE |
|         Real |         C_REAL |
|     units |     C_ATTRIBUTE |
|         CODED_SIMPLE |         C_COMPLEX_OBJECT |
|             value |             C_ATTRIBUTE |
|                 String |                 C_STRING |

Since the Archetype Object Model is also used to constrain other reference models, as for example the openEHR Reference Model, there will be a need to transform openEHR archetypes to ISO 13606 archetypes, and vice versa. The openEHR Reference Model also uses the same primitive datatypes, but includes a different set of complex datatypes, such as DV_ORDINAL, or DV_TEXT[1]. When transforming an openEHR archetype constraint to an ISO 13606 archetype, it might be necessary to introduce an additional CLUSTER structure to represent the equivalent openEHR sub-components as ELEMENTs.

For example, a representation of an openEHR DV_ORDINAL in ISO 13606 would have the structure shown in Table 2.

**Table 2 — Example structure for representing an ordinal data value**

| **openEHR** | **ISO 13606** |
|---|---|
| DV_ORDINAL | CLUSTER matches { -- DV_ORDINAL |
| |     parts matches { |
|     symbol |         ELEMENT matches { -- symbol |
| |             value matches { |
|         DV_CODED_TEXT |                 CODED_VALUE matches {*} |
| |             } |
| |         } |
|     value |         ELEMENT matches { -- value |
| |             value matches { |
|         Integer |                 INTEGER matches {*} |
| |             } |
| |         } |
| |     } |
| | } |

An example of how the LINK class defined in Part 1 of this standard series can be represented using the Archetype Object Model defined in this document is given in Annex B.

1) Please see http://www.openehr.org/releases/RM/latest/docs/data_types/data_types.html#_text_package for the specification of this datatype.

## 0.3 Archetype repositories

The range of archetypes required within a shared EHR community will depend upon its range of clinical activities. The total set needed on a national basis is presently unknown, but there might eventually be several thousand archetypes globally. The ideal sources of knowledge for developing such archetype definitions will be clinical guidelines, care pathways, scientific publications and other embodiments of best practice. However, “de facto” sources of agreed clinical data structures might also include:

— the data schemata (models) of existing clinical systems;

— the lay-out of computer screen forms used by these systems for data entry and for the display of analyses performed;

— data-entry templates, pop-up lists and look-up tables used by these systems;

— shared-care data sets, messages and reports used locally and nationally;

— the structure of forms used for the documentation of clinical consultations or summaries within paper records;

— health information used in secondary data collections;

— the pre-coordinated terms in terminology systems.

Despite this list of *de facto* ways in which clinical data structures are currently represented, these formats are very rarely interoperable without substantial costs. The use of standardised archetypes provides an interoperable way of representing and sharing these specifications, in support of consistent (good quality) health care record-keeping and the semantic interoperability of shared EHRs.

The involvement of national health services, academic organisations and professional bodies in the development of archetypes will enable this approach to contribute to the pursuit of quality evidence-based clinical practice. A key next challenge is to foster communities to build up libraries of archetypes. It is beyond the scope of this document to assert how this work should be advanced, but in several countries so far it would appear that national eHealth programmes are beginning to organise clinical-informatics-vendor teams to develop and operationalise sets of archetypes to meet the needs of specific healthcare domains. In the future regional or national public domain libraries of archetype definitions might be accessed via the Internet, and downloaded for local use within EHR systems. Such usage will also require processes to verify and certify the quality of shared archetypes, which are also beyond the scope of this document but are being taken forward by not for profit organisations such as the open EHR Foundation (www.openehr.org), the Clinical Information Modeling Initiative (CIMI, http://www.opencimi.org) the EN13606 Association (http://www.en13606.org) and the European Institute for Innovation through Health Data (www.i-hd.eu).

## 0.4 Communicating archetypes

This document specifies, in Clause 6, the requirements for a comprehensive and interoperable archetype representation and defines, in Clause 7, the ODP Information Viewpoint representation for the Archetype Object Model.

This document does not require that any particular model be adopted as the internal architecture of archetype repositories, services or components used to author, store or deploy archetypes in collaboration with EHR services. It does require that these archetypes are capable of being mapped to the Archetype Object Model defined in this document in order to support EHR communication and interoperability within an EHR-sharing community.

A more detailed overview of archetypes can be found here:

http://www.openehr.org/releases/AM/latest/docs/Overview/Overview.html

# Health informatics — Electronic health record communication —

# Part 2: Archetype interchange specification

## 1 Scope

This document specifies a means for communicating part or all of the electronic health record (EHR) of one or more identified subjects of care between EHR systems, or between EHR systems and a centralised EHR data repository.

It can also be used for EHR communication between an EHR system or repository and clinical applications or middleware components (such as decision support components) that need to access or provide EHR data, or as the representation of EHR data within a distributed (federated) record system.

This document will predominantly be used to support the direct care given to identifiable individuals, or to support population monitoring systems such as disease registries and public health surveillance. Uses of health records for other purposes such as teaching, clinical audit, administration and reporting, service management, research and epidemiology, which often require anonymization or aggregation of individual records, are not the focus of this standard series but such secondary uses might also find it useful.

This document defines an Archetype Model to be used to represent Archetypes when communicated between repositories, and between archetype services. It defines an optional serialised representation, which may be used as an exchange format for communicating individual archetypes. Such communication might, for example, be between archetype libraries or between an archetype service and an EHR persistence or validation service.

## 2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

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

ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*

ISO 13606-1, *Health informatics — Electronic health record communication — Part 1: Reference model*

## 3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO 13606-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

**3.1**
**archetype repository**
persistent repository of archetype definitions, accessed by a client authoring tool or by a run-time component within an electronic health record service

**3.2**
**concept**
unit of knowledge created by a unique combination of characteristics

[SOURCE: ISO 1087-1:2000]

Note 1 to entry: Concepts are not necessarily bound to particular languages. They are, however, influenced by the social or cultural background often leading to different categorizations.

**3.3**
**operational template**
template in which all references have been substituted by the corresponding structure

**3.4**
**template**
archetype defining a particular document or message intended for specific use cases

## 4 Abbreviations

For the purposes of this document, the following abbreviations apply.

| | |
|---|---|
| ADL | Archetype Definition Language |
| AOM | Archetype Object Model (synonym for Archetype Model) |
| CIMI | Clinical Information Modelling Initiative |
| EHR | Electronic Health Record |
| ODP | Open Distributed Processing (ISO/IEC 10746 series, used for describing distributed systems) |
| OWL | Ontology Web Language |
| RM | Reference Model e.g. the ISO 13606 Part 1 Reference Model |
| UML | Unified Modelling Language |
| XML | Extensible Mark-up Language |

## 5 Conformance

The communication of an archetype that is used to constrain part of an EHR_EXTRACT shall conform to the information model defined in Clause 7. Conformance to the functions defined for each class in Clause 7, where specified, is optional. This document does not prescribe any particular representation of archetypes to be used internally within an archetype repository, server or EHR system. The representation of archetypes shall meet the requirements listed in Clause 6.

# 6 Archetype representation requirements

## 6.1 General

This clause lists a set of formal requirements for an archetype representation. This provides the basis on which the archetype model specified in 7.2 has been designed.

## 6.2 Archetype definition, description and publication information

### 6.2.1 The definition of an archetype shall include the following information.

**6.2.1.1** The globally-unique identifier of this archetype definition.

**6.2.1.2** The identifier of the repository in which this archetype originated or is now primarily held, or of the authority responsible for maintaining it. This repository shall be the one in which the definitive publication status of this archetype will be managed.

**6.2.1.3** The concept that best defines the overall clinical scope of instances conforming to this archetype as a whole, expressed as a coded term or as free text in a given natural language.

**6.2.1.4** The health informatics domain to which this archetype applies (e.g. EHR). This shall map to a set of reference models with which this archetype may be used.

**6.2.1.5** The underlying reference model for which this archetype was ideally fashioned.

NOTE An archetype can be capable of use with more than one relevant reference model within a given health informatics domain, but it is expected that the archetype will be optimised for one.

**6.2.1.6** The natural language in which this archetype was originally defined, represented by its ISO 639-code. In the event of imprecise translations, this is the definitive language for interpretation of the archetype.

### 6.2.2 The definition of an archetype may include the following information, if applicable.

**6.2.2.1** The globally-unique identifier for the archetype of which this archetype is a specialisation and to which it shall also conform.

**6.2.2.2** The globally-unique identifier of the former archetype that this definition replaces, if it not the first version of an archetype.

**6.2.2.3** The reason for defining this new version of a pre-existing archetype.

**6.2.2.4** The identifier of the replacement for this archetype, if it has been superseded.

NOTE It is possible that this information can only be added by reference within a version-controlled repository; how this is effected is not in scope for this document.

**6.2.2.5** An archetype shall have one or more description sets, defining its usage and purpose. Multiple versions of this information may be included, represented in different natural languages or to inform different kinds of potential user.

**6.2.3 An archetype description set shall include the following information.**

**6.2.3.1** The uniquely-identified person or organisation responsible for providing this description set. This may include contact information for that person or organisation.

**6.2.3.2** The uniquely-identified person or organisation responsible for defining the archetype hierarchy itself. This may include contact information for that person or organisation.

**6.2.3.3** The natural language in which this description set is provided, represented by its ISO 639-code.

**6.2.3.4** A formal statement defining the scope and purpose of this archetype, expressed as a coded term or as free text in a given natural language.

NOTE These criteria can be expressed as coded terms to improve queries for relevant archetypes from the repository.

EXAMPLE The scope and purpose can specify:

1) the principal clinical specialty or kinds of user for which it is intended;

2) A list of clinical terms (keywords): diagnoses, acts, drugs, findings etc.;

3) the kind of patient in whom it is intended to be used (age, gender, etc.);

4) the kind of demographic entities it is intended to represent.

**6.2.4 An archetype description set may include the following information, if applicable.**

**6.2.4.1** A formal statement of the intended use of this archetype.

NOTE Ideally this can be a coded expression, although a suitable terminology for this is not yet available.

**6.2.4.2** A formal statement of situations in which users might erroneously believe this archetype should be used. This may also stipulate any kinds of Reference Model for which it is unsuitable.

**6.2.4.3** A detailed explanation of the purpose of this archetype, including any features of particular interest or note. This may include an indication of the persons for which this definition is intended e.g. for students. This information might be included explicitly, and/or by reference (e.g. via a URL).

**6.2.4.4** A description, reference or link to the published medical knowledge that has underpinned the definition of this archetype.

**6.2.4.5** Information about evidence that has informed its development, e.g. an existing specification or standard, published knowledge or clinical experience.

**6.2.4.6** How the archetype may be used in quality healthcare delivery.

**6.2.4.7** The care processes it has been designed to support.

**6.2.4.8** Information about which organisations, professional bodies or government bodies have endorsed the model, when this endorsement occurred, and under which criteria.

**6.2.5 An archetype definition shall include a statement of its publication status.**

**6.2.5.1** An archetype definition may evolve through a series of publication states, for example an approval process, without otherwise being changed. These successive states shall be retained as part of the archetype, for audit purposes. However, the modification of the publication status of an archetype shall not itself constitute a formal revision of the identifier by which the archetype is referenced within an EHR_EXTRACT, since the constraint specification will not have been changed.

**6.2.6 The publication status of an archetype shall specify the following information.**

**6.2.6.1** The publication status of this archetype, taken from the following list:

— Test;

— In development;

— Release candidate;

— Rejected;

— Definitive;

— Deprecated.

**6.2.6.2** The date when this particular publication status applied

NOTE The first instance of a publication status for this archetype will also be the date when it was first composed.

**6.2.6.3** The unique identifier of the person committing this archetype to the repository and thereby asserting this publication status. This identification might optionally include the organisation which that person represents.

**6.2.6.4** The unique identifier of the body authorising this change in publication status.

**6.2.6.5** The date when it is anticipated that the present publication status, and the archetype content itself, ought to be reviewed to confirm it remains valid.

**6.2.6.6** The unique identifier of the person or organisation that is nominated, authorised or has accepted responsibility for reviewing the validity of the archetype and optionally for updating it, when appropriate.

**6.2.6.7** A clear statement of any copyright or licensing restrictions which apply to the use of the archetype.

**6.2.6.8** The copyright holder and/or governing authority.

**6.2.7 Version management.**

**6.2.7.1** An archetype definition shall indicate the version of the constraints it specifies.

**6.2.7.2** An archetype definition may indicate the person or organization responsible for that version.

**6.2.7.3** An archetype definition may indicate the date on which the current version was created.

**6.2.7.4** The archetype version identifiers or other properties may indicate the nature of changes made from the previous version, and in particular if EHR instances communicated with the current and the previous version are compatible with each other.

## 6.3 Archetype node constraints

### 6.3.1 General

An archetype definition shall include a specification of the hierarchical schema to which instances of data (e.g. EHR data) shall conform. This schema defines the hierarchical organisation of a set of nodes, the relationships between them, and constraints on the permitted values of attributes and data values. These shall also conform to the underlying reference model(s) for which this definition is applicable.

### 6.3.2 Archetype node references

**6.3.2.1** Any node in the archetype hierarchy might be defined explicitly or, by reference, be specified to be part or whole of a pre-existing archetype.

**6.3.2.2** A reference to a pre-existing archetype or archetype fragment may be explicit, by specifying the archetype identifier, and optionally the identifier of the node of the archetype fragment.

**6.3.2.3** A reference to an archetype fragment may be internal to (i.e. part of) the current archetype.

**6.3.2.4** An archetype node may be specified to be one of a set of possible archetypes, by defining an explicit list of candidates and/or by specifying a set of constraints on any of the attributes of an archetype definition.

**6.3.2.5** In addition to specifying one or more archetype fragments by reference or constraint, it shall be possible to include an explanation of the rationale for incorporating that specification at the given point in the current archetype hierarchy.

### 6.3.3 The specification of an archetype node (if not by reference) shall include the following information.

**6.3.3.1** A unique identifier of this archetype node. Either in itself or when combined with the globally-unique identifier of this archetype definition, it shall be a globally unique reference to the node itself.

**6.3.3.2** The class in the EHR instance hierarchy, mapping to the underlying reference model that this archetype constrains, that shall be instantiated in order to conform to this archetype node.

**6.3.3.3** The number of occurrences, expressed as a range that may be instantiated corresponding to this archetype node within an instance hierarchy.

**6.3.3.4** Other constraints and rules may optionally be specified to govern the creation of instances corresponding to this archetype node.

**6.3.3.5** Constraint rules may be expressed as logical conditions, and may include reference to environment parameters such as the current time or location or participants, or be related to the (pre-existing) values other nodes in the instance hierarchy. Constraint rules may be used to represent the relationship between EHR data and workflow or care pathway processes.

**6.3.3.6** Constraint rules may be expressed as inclusion or exclusion criteria.

**6.3.3.7** An archetype shall identify the formalism (including version) in which constraint rules are expressed (e.g. ADL, OWL).

### 6.3.4 Binding archetype nodes to terms

**6.3.4.1** Every node of an archetype schema hierarchy shall be associated with at least one term, which most accurately expresses the intended concept to be represented by that node on instantiation in the corresponding instance hierarchy. This term can usually be included or referenced within the instance.

**6.3.4.2** Every node of an archetype schema hierarchy may additionally be associated with a description that elaborates on the meaning given by the term labelling that node.

**6.3.4.3** Any node of an archetype may be mapped to any number of additional concepts, terms and synonyms from terminology systems, to support either the interrogation of the archetype repository or of the corresponding instances.

**6.3.4.4** Any concept mapping term or text shall specify the purpose that this mapping serves, using one of the following list of values:

— Principal concept;

— Term binding;

— Synonym;

— Language translation.

**6.3.4.5** Any reference to a coded term shall include the code, rubric, and identify the coding system (including version) from which the code and rubric have been taken. In addition, it shall be possible to specify the natural language in which this term was mapped, or in which a translation is expressed.

**6.3.4.6** The meaning of every node in an archetype should be defined through a binding to the appropriate concept identifier from an appropriate concept model.

**6.3.4.7** The meaning of any specified link between two or more nodes in an archetype may be defined through a binding to the appropriate concept identifier from an appropriate concept model.

### 6.3.5 Attribute and association constraints

**6.3.5.1** An archetype node may specify constraints on any attributes or associations that correspond to the attributes and associations of that node in the underlying reference model.

NOTE These constraints can pre-determine or restrict some or all of the contextual information that is included within the corresponding instance, as represented within the reference model. Context information, such as the person to whom a particular observation or inference relates, is formally-represented in most generic EHR-like models to facilitate safe querying and retrieval, even if that information can be inferred from the archetype name or an axis within a terminology system used for the data value. Some archetypes or fragments will pre-determine the values of some of these, which shall be capable of specification within the archetype definition (for example, to constrain the subject of information to be a relative of the patient, and not the patient, in an archetype for family history).

### 6.3.6 For any given reference model property it shall be possible to specify the following information.

**6.3.6.1** The name of the attribute or association, mapping to the underlying reference model for which this archetype was ideally fashioned, to which this constraint applies.

**6.3.6.2** For a given reference model, if an attribute or association corresponding to this aspect of context is mandatory to be included within a valid EHR instance.

**6.3.6.3** The number of instances (expressed as a range) corresponding to this aspect of context that may be instantiated.

**6.3.6.4** If multiple instances are permitted, it shall be possible to specify if these are to be represented as an ordered or unordered list.

**6.3.6.5** If multiple instances are permitted, it shall be possible to specify if the corresponding data values (of leaf nodes or attributes) shall be unique.

**6.3.6.6** Constraints may be specified for the data values of leaf nodes or leaf attributes.

**6.3.6.7** It shall be possible to specify if instances conforming to an archetype shall include one or more optional or mandatory LINKs, where the value of the *role* attribute is to be one of a specified set of codes, and where the *target* attribute should refer to an instance of a RECORD_COMPONENT based on an archetype that is one of a specified set of archetypes.

**6.3.6.8** It shall be possible to specify that the LINK shall have an external target where the *target information type* should be included in a specified set of such types.

**6.3.6.9** Other constraints and rules may optionally be specified to govern the creation of instances corresponding to a reference model attribute or association.

## 6.4 Data value constraints

**6.4.1** It shall be possible to specify constraints and rules for the data values of leaf nodes in the Reference Model hierarchy, or for any other attributes of any archetype node.

**6.4.2** Constrains on a leaf node shall include specifying a single datatype for instance values, in conformance with the underlying reference model that this archetype constrains.

**6.4.3** It shall be possible to specify the following data value constraint information.

**6.4.3.1** If the data value is permitted to have a null value, and optionally to specify a reason (e.g. to specify a null flavour value).

**6.4.3.2** If the constraint or rule is an inclusion or exclusion criterion.

**6.4.3.3** The formalism (including version) in which this constraint specification is represented.

**6.4.3.4** The intended fixed (prescribed) value for conforming instances.

**6.4.3.5** The intended default value for conforming instances.

**6.4.3.6** A list of permitted candidate values for conforming instances (i.e. to be a subset of those values legally permissible in the underlying reference model).

**6.4.4** For quantity datatypes it shall be possible to specify:

— a value range within which values for conforming instances shall lie;

— a range within which values are considered exceptional or critical (e.g. cut off points, ranges, min, max values);

— the intended measurement units for conforming instances.

**6.4.5** For date and time datatypes it shall be possible to specify:

— a value range within which values for conforming instances shall lie;

— the intended measurement units for conforming instances.

**6.4.6** For textual datatypes it shall be possible to specify:

— a string pattern defining a range of possible values;

— the intended coding scheme to be used for conforming instances;

— a valid value domain, by referencing a list of individual concepts from a given terminology (with appropriate identification of the terminology referenced);

— a valid value domain, by stating an intentional definition of the set of valid values from an identified terminology system;

— a valid value domain, by binding the coded data element to a predefined reference set from an identified terminology system;

— terminology value domains from different languages;

— optionally to specify different terminology value domains for different documentation purposes.

**6.4.7** Constraint rules might be expressed as logical conditions, and may include reference to environment parameters such as the current time or location or participants, or be related to the (pre-existing) values other nodes in the instance hierarchy.

**6.4.8** The reference to a pre-existing value shall specify that instance precisely and unambiguously. For example, it might be necessary to include a reference to:

— the archetype identifier;

— the archetype node identifier;

— the attribute or association name;

— the occurrence in the instance hierarchy, for example:

  — first;

  — most-recent;

  — any;

  — n ordered by y (the nth element of a set of instances ordered on y);

  — highest value;

  — lowest value;

  — one or more instances within a (definable) recent time interval.

— the intended relationship between this specified instance value and the data value being constrained, for example:

  — the same value as;

  — a subset or substring of;

  — greater than, greater than or equal to, less than, less than or equal to;

— earlier than, later than, etc.;

— if ... then...;

— shall not be the same as.

**6.4.9** These relative constraints may be nested, and include logical or set operators in order to represent compound rules.

# 7 Archetype object model

## 7.1 Preface

### 7.1.1 Purpose

This clause contains the normative description of archetype and template semantics in the form of an object model. The model presented here can be used as a basis for building software that represents archetypes and templates, independent of their persistent representation. Equally, it can be used to develop the output side of parsers that process archetypes in a linguistic format.

### 7.1.2 Nomenclature

In this document, the term 'attribute' denotes any stored property of a type defined in an object model, including primitive attributes and any kind of relationship such as an association or aggregation. XML 'attributes' are always referred to explicitly as 'XML attributes'.

The word 'archetype' in a broad sense is also used to designate what are commonly understood to be 'archetypes' (specifications of clinical data groups / data constraints) and 'templates' (data sets based on archetypes). Statements about 'archetypes' in this specification can be always understood to also apply to templates, unless otherwise indicated.

## 7.2 Model overview

The model described here is a pure object-oriented model that can be used with archetype parsers and software that manipulates archetypes and templates in memory. It is typically the output of a parser of any serialised form of archetypes.

### 7.2.1 Package structure

The Archetype Object Model is defined as the package `am.archetype` , as illustrated in Figure 1.

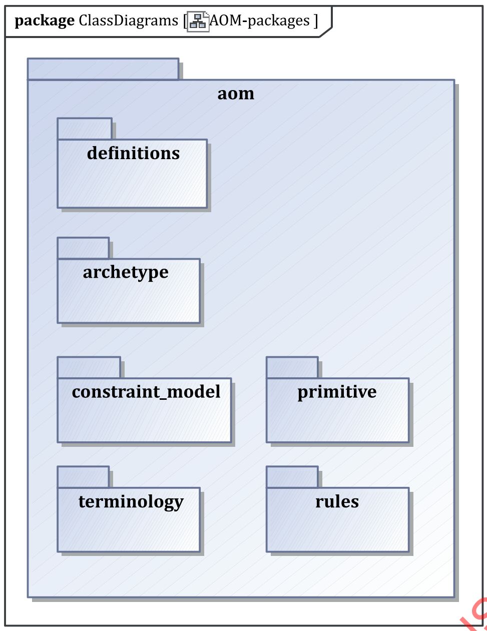


Figure 1 — Package Overview

### 7.2.2 Definition and utility classes

#### 7.2.2.1 Overview

Various definitional classes are used in the AOM. Some are defined in the `aom.definitions` package, while others come from the `definitions` package. These are illustrated in Figure 2 below.

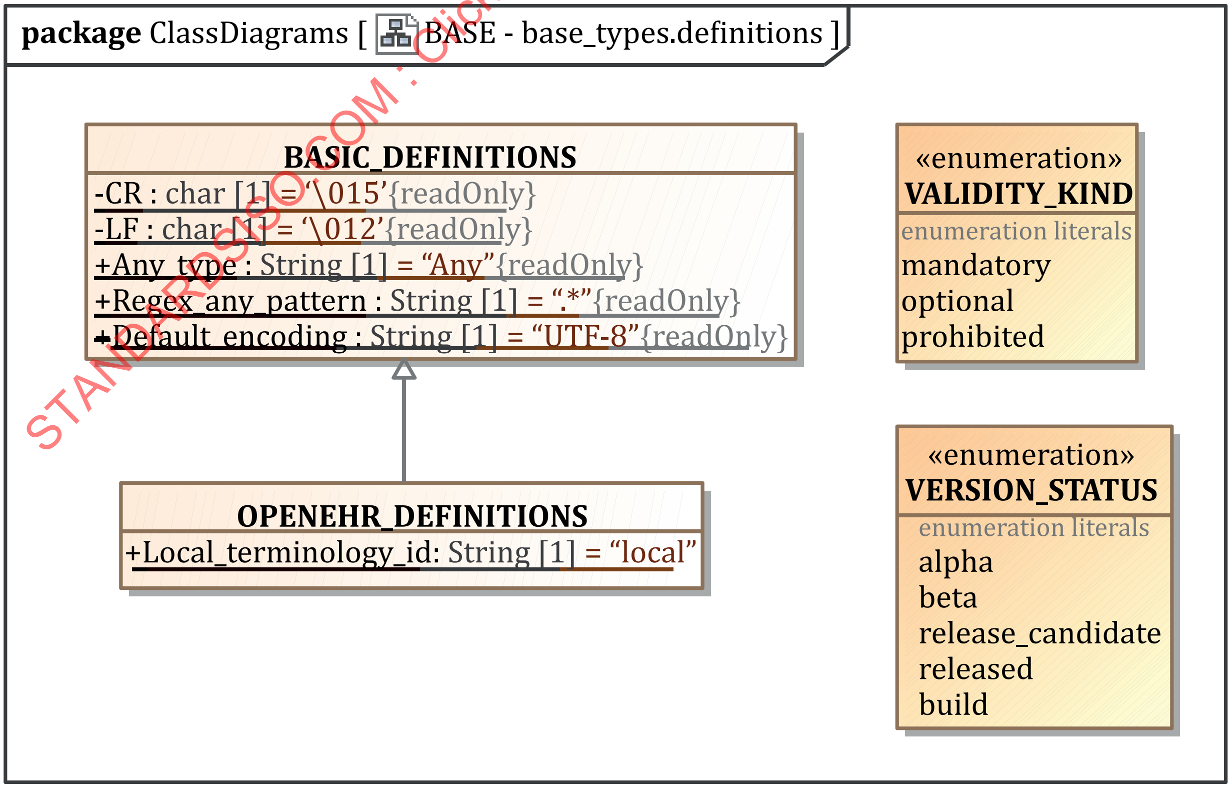

package ClassDiagrams [ BASE - base_types.definitions ]
BASIC_DEFINITIONS
-CR : char [1] = '\015'{readOnly}
-LF : char [1] = '\012'{readOnly}
+Any_type : String [1] = "Any"{readOnly}
+Regex_any_pattern : String [1] = ".*"{readOnly}
+Default_encoding : String [1] = "UTF-8"{readOnly}
OPENEHR_DEFINITIONS
+Local_terminology_id: String [1] = "local"
«enumeration»
VALIDITY_KIND
enumeration literals
mandatory
optional
prohibited
«enumeration»
VERSION_STATUS
enumeration literals
alpha
beta
release_candidate
released
build

Figure 2 — Definition Package

The enumeration type `VALIDITY_KIND` is provided in order to define standard values representing `mandatory`, `optional`, or `disallowed` in any model. It is used in this model in classes such as `C_DATE` , `C_TIME` and `C_DATE_TIME`. The `VERSION_STATUS` enumeration type serves a similar function within various AOM types.

Other classes used from the `base` package include `AUTHORED_RESOURCE` (`resource` package) and its subordinate classes. These are shown in full within the packages that use them.

#### 7.2.2.2 Class definitions

##### 7.2.2.2.1 VERSION_STATUS enumeration

| **Enumeration** | **VERSION_STATUS** | |
|---|---|---|
| **Description** | Status of a versioned artefact, as one of a number of possible values: uncontrolled, prerelease, release, build. | |
| **Attributes** | **Signature** | **Meaning** |
| | alpha | Value representing a version which is 'unstable', i.e. contains an unknown size of change with respect to its base version. Rendered with the build number as a string in the form "N.M.P-alpha.B" e.g. "2.0.1-alpha.154". |
| | beta | Value representing a version which is 'beta', i.e. contains an unknown but reducing size of change with respect to its base version. Rendered with the build number as a string in the form "N.M.P-beta.B" e.g. "2.0.1-beta.154". |
| | release_candidate | Value representing a version which is 'release candidate', i.e. contains only patch-level changes on the base version. Rendered as a string as "N.M.P-rc.B" e.g. "2.0.1-rc.27". |
| | released | Value representing a version which is 'released', i.e. is the definitive base version. Rendered with the build number as a string in the form "N.M.P" e.g. "2.0.1". |
| | build | Value representing a version which is a build of the current base release. Rendered with the build number as a string in the form "N.M.P+B" e.g. "2.0.1+33". |

##### 7.2.2.2.2 VALIDITY_KIND enumeration

| **Enumeration** | **VALIDITY_KIND** | |
|---|---|---|
| **Description** | An enumeration of three values that might commonly occur in constraint models.<br>Use as the type of any attribute within this model, which expresses constraint on some attribute in a class in a reference model. For example to indicate validity of Date/Time fields. | |
| **Attributes** | **Signature** | **Meaning** |
| | mandatory | Constant to indicate mandatory presence of something. |
| | optional | Constant to indicate optional presence of something. |
| | prohibited | Constant to indicate disallowed presence of something. |

##### 7.2.2.2.3 ADL_CODE_DEFINITIONS class

| **Class** | **ADL_CODE_DEFINITIONS** | |
|---|---|---|
| **Description** | Definitions relating to the internal code system of archetypes. | |
| **Attributes** | **Signature** | **Meaning** |

| | | |
|---|---|---|
| **1..1** | **Id_code_leader**: `String = "id"` | String leader of 'identifier' codes, i.e. codes used to identify archetype nodes. |
| **1..1** | **Value_code_leader**: `String = "at"` | String leader of 'value' codes, i.e. codes used to identify codes values, including value set members. |
| **1..1** | **Value_set_code_leader**: `String = "ac"` | String leader of 'value set' codes, i.e. codes used to identify value sets. |
| **1..1** | **Specialisation_separator**: `char = '.'` | Character used to separate numeric parts of codes belonging to different specialisation levels. |
| **1..1** | **Code_regex_pattern**: `String = "(0\|[1-9][0-9]*)(\.(0\|[1-9][0-9]*))*"` | Regex used to define the legal numeric part of any archetype code. Corresponds to the simple pattern of dotted numbers, as used in typical multi-level numbering schemes. |
| **1..1** | **Root_code_regex_pattern**: `String = "^id1(\.1)*$"` | Regex pattern of the root id code of any archetype. Corresponds to codes of the form id1, id1.1, id1.1.1 etc.. |
| **1..1** | **Primitive_node_id**: `String = "id9999"` | Code id used for C_PRIMITIVE_OBJECT nodes on creation. |

## 7.3 The archetype package

### 7.3.1 Overview

The top-level model of archetypes and templates (all variant forms) is illustrated in Figure 3: Archetype Package. The model defines a standard structural representation of an archetype. Archetypes authored as independent entities are instances of the class `AUTHORED_ARCHETYPE` which is a descendant of `AUTHORED_RESOURCE` and `ARCHETYPE`. The former provides a standardised model of descriptive meta-data, language information, annotations and revision history for any resource. The latter class defines the core structure of any kind of archetype, including definition, terminology, and optional rules part, along with a 'semantic identifier' (`ARCHETYPE.archetype_id`).

The `AUTHORED_ARCHETYPE` class adds identifying attributes, flags and descriptive meta-data, and is the ancestor type for two further specialisations - `TEMPLATE` and `OPERATIONAL_TEMPLATE`. The `TEMPLATE` class defines the notion of a 'templated' archetype, i.e. an archetype containing fillers/references (e.g. ADL's `use_archetype` statements), typically designed to represent a data set. To enable this, it may contain 'overlays', private archetypes that specialise one or more of the referenced / filler archetypes it uses. Overlays are instances of the `TEMPLATE_OVERLAY` class, have no meta-data of their own, but are otherwise computationally just like any other archetype.

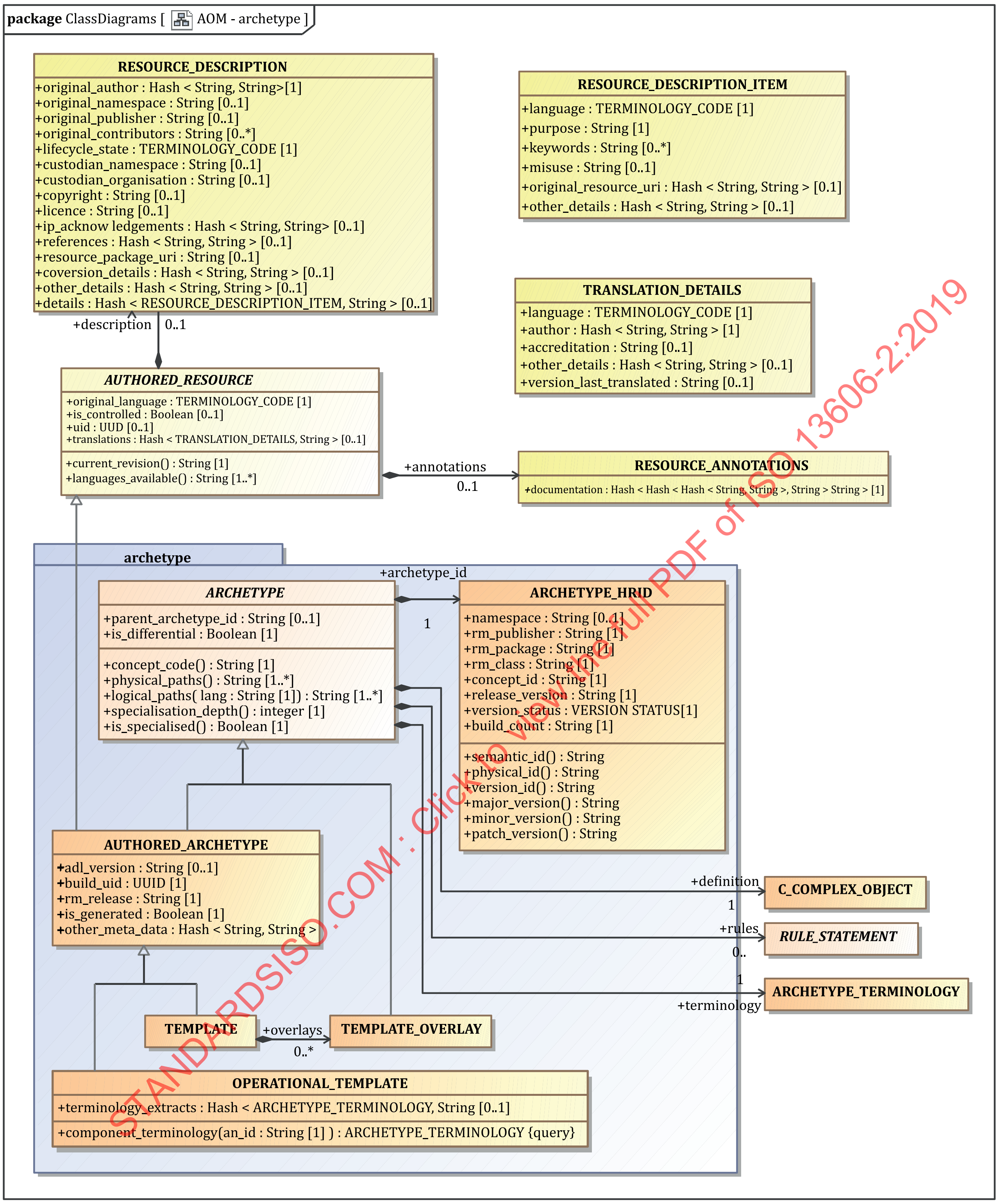

package ClassDiagrams [ AOM - archetype ]
RESOURCE_DESCRIPTION
+original_author : Hash < String, String>[1]
+original_namespace : String [0..1]
+original_publisher : String [0..1]
+original_contributors : String [0..*]
+lifecycle_state : TERMINOLOGY_CODE [1]
+custodian_namespace : String [0..1]
+custodian_organisation : String [0..1]
+copyright : String [0..1]
+licence : String [0..1]
+ip_acknow ledgements : Hash < String, String> [0..1]
+references : Hash < String, String > [0..1]
+resource_package_uri : String [0..1]
+coversion_details : Hash < String, String > [0..1]
+other_details : Hash < String, String > [0..1]
+details : Hash < RESOURCE_DESCRIPTION_ITEM, String > [0..1]
RESOURCE_DESCRIPTION_ITEM
+language : TERMINOLOGY_CODE [1]
+purpose : String [1]
+keywords : String [0..*]
+misuse : String [0..1]
+original_resource_uri : Hash < String, String > [0.1]
+other_details : Hash < String, String > [0..1]
TRANSLATION_DETAILS
+language : TERMINOLOGY_CODE [1]
+author : Hash < String, String > [1]
+accreditation : String [0..1]
+other_details : Hash < String, String > [0..1]
+version_last_translated : String [0..1]
+description
0..1
AUTHORED_RESOURCE
+original_language : TERMINOLOGY_CODE [1]
+is_controlled : Boolean [0..1]
+uid : UUD [0..1]
+translations : Hash < TRANSLATION_DETAILS, String > [0..1]
+current_revision() : String [1]
+languages_available() : String [1..*]
+annotations
0..1
RESOURCE_ANNOTATIONS
+documentation : Hash < Hash < Hash < String, String >, String > String > [1]
archetype
+archetype_id
1
ARCHETYPE
+parent_archetype_id : String [0..1]
+is_differential : Boolean [1]
+concept_code() : String [1]
+physical_paths() : String [1..*]
+logical_paths( lang : String [1]) : String [1..*]
+specialisation_depth() : integer [1]
+is_specialised() : Boolean [1]
ARCHETYPE_HRID
+namespace : String [0..1]
+rm_publisher : String [1]
+rm_package : String [1]
+rm_class : String [1]
+concept_id : String [1]
+release_version : String [1]
+version_status : VERSION STATUS[1]
+build_count : String [1]
+semantic_id() : String
+physical_id() : String
+version_id() : String
+major_version() : String
+minor_version() : String
+patch_version() : String
AUTHORED_ARCHETYPE
+adl_version : String [0..1]
+build_uid : UUID [1]
+rm_release : String [1]
+is_generated : Boolean [1]
+other_meta_data : Hash < String, String >
+definition
1
C_COMPLEX_OBJECT
+rules
0..
RULE_STATEMENT
1
+terminology
ARCHETYPE_TERMINOLOGY
TEMPLATE
+overlays
0..*
TEMPLATE_OVERLAY
OPERATIONAL_TEMPLATE
+terminology_extracts : Hash < ARCHETYPE_TERMINOLOGY, String [0..1]
+component_terminology(an_id : String [1] ) : ARCHETYPE_TERMINOLOGY {query}

**Figure 3 — Archetype Package**

The `OPERATIONAL_TEMPLATE` class represents the fully flattened form of a template, i.e. with all fillers and references substituted and overlays processed, to form what is in practical terms, a single custom-made 'operational' artefact, ready for transformation to downstream artefacts. Because an operational template includes one or more other archetype structures inline, it also includes their terminologies, enabling it to be treated as a self-standing artefact.

### 7.3.2 Archetype identification

#### 7.3.2.1 Human-Readable Identifier (HRID)

All archetype variants based on ARCHETYPE have a human-readable, structured identifier defined by the ARCHETYPE_HRID class. This identifier places the artefact in a multi-dimensional space based on a namespace, its reference model class and its informational concept. This class defines an atomised representation of the identifier, enabling variant forms to be used as needed. Its various parts can be understood from Figure 4, which also shows the computed semantic_id and physical_id forms.

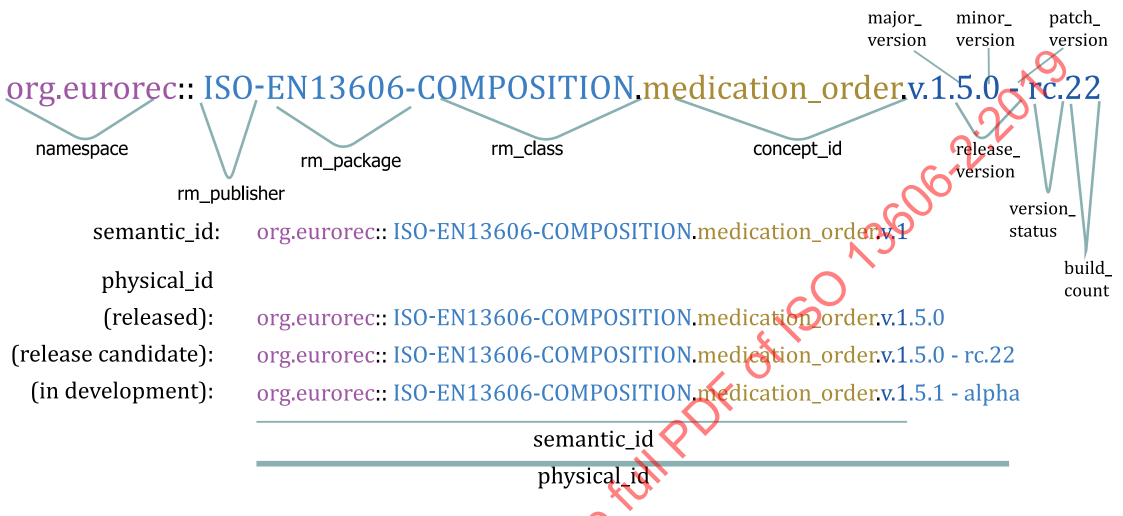

major_ version
minor_ version
patch_ version
org.eurorec:: ISO-EN13606-COMPOSITION.medication_order.v.1.5.0 - rc.22
namespace
rm_publisher
rm_package
rm_class
concept_id
release_ version
version_ status
build_ count
semantic_id:
org.eurorec:: ISO-EN13606-COMPOSITION.medication_order.v.1
physical_id
(released):
org.eurorec:: ISO-EN13606-COMPOSITION.medication_order.v.1.5.0
(release candidate):
org.eurorec:: ISO-EN13606-COMPOSITION.medication_order.v.1.5.0 - rc.22
(in development):
org.eurorec:: ISO-EN13606-COMPOSITION.medication_order.v.1.5.1 - alpha
semantic_id
physical_id

**Figure 4 — Archetype HRID structure**

For specialised archetypes, the parent_archetype_id is also required. This is a string reference to an archetype, and is normally the 'interface' form of the id, i.e. down to the major version only. In some circumstances, it is useful to include the minor and patch version numbers as well.

An important aspect of identification relates to the rules governing when the HRID namespace changes or is retained, with respect to when 'moves' or 'forks' occur. Its value is always the same as one of the original_namespace and custodian_namespace properties inherited from AUTHORED_RESOURCE. description (or both, in the case where they are the same).

#### 7.3.2.2 Machine identifiers

Two machine identifiers are defined for archetypes. The ARCHETYPE.uid attribute defines a machine identifier equivalent to the human readable archetype_id.semantic_id, i.e. ARCHETYPE_HRID up to its major version, and changes whenever the latter does. It is defined as optional but to be practically useful would need to be mandatory for all archetypes within a custodian organisation where this identifier was in use. It could in principle be synthesised at any time for a custodian that decided to implement it.

The ARCHETYPE.build_uid attribute is also optional, and if used, is intended to provide a unique identifier that corresponds to any change in version of the artefact. At a minimum, this means generating a new UID for each change to:

— ARCHETYPE.*archetype_id.release_version*;

— ARCHETYPE.*archetype_id.build_count*;

— ARCHETYPE.*description.lifecycle_state*.

For every change made to an archetype inside a controlled repository (for example, addition or update of meta-data fields), this field should be updated with a new GUID value, generated in the normal way.

### 7.3.3 Top-level meta-data

#### 7.3.3.1 ADL version

The version of the archetype formalism in which the current archetype is expressed. For reasons of convenience, the version number is still taken from the ADL specification, but now refers to all archetype-related specifications together, since they are always updated in a synchronised fashion.

#### 7.3.3.2 Reference model release

The `ARCHETYPE.rm_release` attribute designates the release of the reference model on which the archetype is based, in the archetype's current version. This means `rm_release` can change with new versions of the archetype, where re-versioning includes upgrading the archetype to a later RM release. However, such upgrading still has to obey the basic rule of archetype compatibility: later minor, patch versions and builds cannot create data that is not valid with respect to the prior version.

This should be in the same semver.org 3-part form as the `ARCHETYPE_HRID.release_version` property, e.g. "1.0.2". This property does not indicate conformance to any particular reference model version(s) other than the named one, since most archetypes can easily conform to more than one. More minimal archetypes are likely to technically conform to more old and future releases than more complex archetypes.

#### 7.3.3.3 Generated flag

The `ARCHETYPE.is_generated` flag is used to indicate that an archetype has been machine-generated from another artefact, e.g. an older ADL version (say 1.4), or a non-archetype artefact. If true, it indicates to tools that the current archetype can potentially be overwritten, and that some other artefact is considered the primary source. If manual authoring occurs, this attribute should be set to false.

### 7.3.4 Governance meta-data

Various meta-data elements are inherited from the `AUTHORED_RESOURCE` class, and provide the natural language description of the archetype, authoring and translation details, use, misuse, keywords and so on. There are three distinct parts of the meta-data: governance, authorship, and descriptive details.

#### 7.3.4.1 Governance meta-data items

Governance meta-data is visible primarily in the `RESOURCE_DESCRIPTION` class, inherited via `AUTHORED_RESOURCE`, and consists of items relating to management and intellectual property status of the artefact.

The optional `resource_package_uri` property enables the recording of a reference to a package of archetypes or other resources, to which this archetype is considered to below. It may be in the form of 'text <URL>'.

**Lifecycle_state**

The `description.lifecycle_state` is an important property of an archetype, which is used to record its state in a defined lifecycle.

**Original_namespace and Original_publisher**

These two optional properties indicate the original publishing organisation, and its namespace, i.e. the original publishing environment where the artefact was first imported or created. The `original_namespace` property is normally the same value as `archetype_id.namespace`, unless the artefact has been forked into its current custodian, in which case `archetype_id.namespace` will be the same as `custodian_namespace`.

**Custodian_namespace and Custodian_organisation**

These two optional properties state a formal namespace, and a human-readable organisation identifier corresponding to the current custodian, i.e. maintainer and publisher of the artefact, if there is one.

**Intellectual property items**

There are three properties in the class that `RESOURCE_DESCRIPTION` relate to intellectual property (IP). Licence is a String field for recording of the licence (US: 'license') under which the artefact can be used. The recommended format is

```
licence name <reliable URL to licence statement>
```

The copyright property records the copyright applying to the artefact, and is normally in the standard form '(c) name' or '(c) year name'. The special character © may also be used (UTF-8 0xC2A9).

### 7.3.4.2 Authorship meta-data

Authorship meta-data consists of items such as author name, contributors, and translator information.

**Original author**

The `RESOURCE_DESCRIPTION.original_author` property defines a simple list of name/value pairs via which the original author can be documented. Typical key values include 'name', 'organi[zs]ation', 'email' and 'date'.

**Contributors**

The `RESOURCE_DESCRIPTION.other_contributors` property is a simple list of strings, one for each contributor. The recommended format of the string is one of:

```
first names last name, organization
first names last name, organisation <contributor email address>
first names last name, organisation <organisation email address>
```

**Languages and translation**

The `AUTHORED_RESOURCE.original_language` and `TRANSLATION_DETAILS` class enable the original language of authoring and information relating to subsequent translations to be expressed. `TRANSLATION_DETAILS.author` allows each translator to be represented in the same way as the `original_author`, i.e. a list of name/values. The `version_last_translated` property is used to record a copy of the archetype_id.physical_id for each language, when the translation was carried out. This enables maintainers to know when new translations are needed for some or all languages.

**Version_last_translated**

This String property records the full version identifier (i.e. `ARCHETYPE.archetype_id.version_id`) at the time of last translation, enabling tools to determine if and when translations might be out of date.

### 7.3.4.3 Descriptive meta-data

Various descriptive meta-data may be provided for an archetype in multiple translations in the `RESOURCE_DESCRIPTION_ITEM` class, using one instance for each translation language.

**Purpose**

The `purpose` item is a String property for recording the intended design concept of the artefact.

**Use and misuse**

The `use` and `misuse` properties enable specific uses and misuses to be documented. The latter normally relate to common errors of use, or apparently reasonable but wrong assumptions about use.

**Keywords**

The `keywords` property is a list of Strings designed to record search keywords for the artefact.

**Resources**

The original_resource_uri property is used to record one or more references to resources in each particular language.

TBD: This property does not appear to have ever been used, and it might not be useful, since 'resources' are not typically available for each language.

## 7.3.5 Structural definition

### 7.3.5.1 Common structural parts

The archetype definition is the main definitional part of an archetype and is an instance of a C_COMPLEX_OBJECT. This means that the root of the constraint structure of an archetype always takes the form of a constraint on a non-primitive object type.

The terminology section of an archetype is represented by its own classes, and is what allows archetypes to be natural language- and terminology-neutral. It is described in detail in the Terminology Package.

An archetype may include one or more rules. Rules are statements expressed in a subset of predicate logic, which can be used to state constraints on multiple parts of an object. They are not needed to constrain single attributes or objects (since this can be done with an appropriate C_ATTRIBUTE or C_OBJECT ), but are necessary for constraints referring to more than one attribute, such as a constraint that 'systolic pressure should be >= diastolic pressure' in a blood pressure measurement archetype. They can also be used to declare variables, including external data query results, and make other constraints dependent on a variable value, e.g. the gender of the record subject.

Lastly, annotations and revision history sections, inherited from the AUTHORED_RESOURCE class, can be included as required. The annotations section is of particular relevance to archetypes and templates, and is used to document individual nodes within an archetype or template, and/or nodes in reference model data, that might not be constrained in the archetype, but whose specific use in the archetyped data needs to be documented. In the former case, the annotations are keyed by an archetype path, while in the latter case, by a reference model path.

### 7.3.5.2 Structural variants

The model in Figure 3: Archetype Package defines the structures of a number of variants of the 'archetype' idea. All concrete instances are instances of one of the concrete descendants of ARCHETYPE. Figure 5: Source Archetype Structure illustrates the typical object structure of a source archetype - the form of archetype created by an authoring tool - represented by a DIFFERENTIAL_ARCHETYPE instance. Mandatory parts are shown in bold.

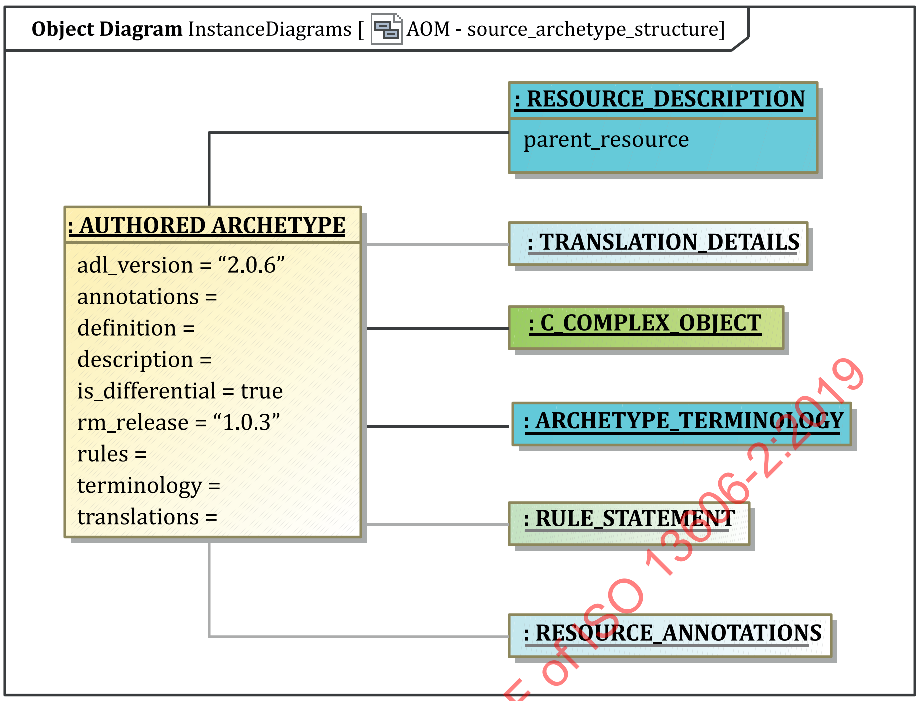


**Figure 5 — Source archetype structure**

Source archetypes can be specialised, in which case their definition structure is a partial overlay on the flat parent, or ‘top-level’, in which case the definition structure is complete. C_ARCHETYPE_ROOT instances may only occur representing direct references to other archetypes - ‘external references’.

A flat archetype is generated from one or more source archetypes via the flattening process described in the next subclause of this specification. This generates a FLAT_ARCHETYPE from a DIFFERENTIAL_ARCHETYPE instance. The main two changes that occur in this operation are a) specialised archetype overlays are applied to the flat parent structure, resulting in a full archetype structure, and b) internal references (use_nodes) are replaced by their expanded form, i.e. a copy of the subtrees to which they point.

This form is used to represent the full ‘operational’ structure of a specialised archetype, and has two uses. The first is to generate backwards compatible ADL 1.4 legacy archetypes (always in flat form); the second is during the template flattening process, when the flat forms of all referenced archetypes and templates are ultimately combined into a single operational template.

Figure 6: Source template structure illustrates the structure of a source template, i.e. instances of TEMPLATE. A source template is an archetype containing C_ARCHETYPE_ROOT objects representing slot fillers - each referring to an external archetype or template, or potentially an overlay archetype.

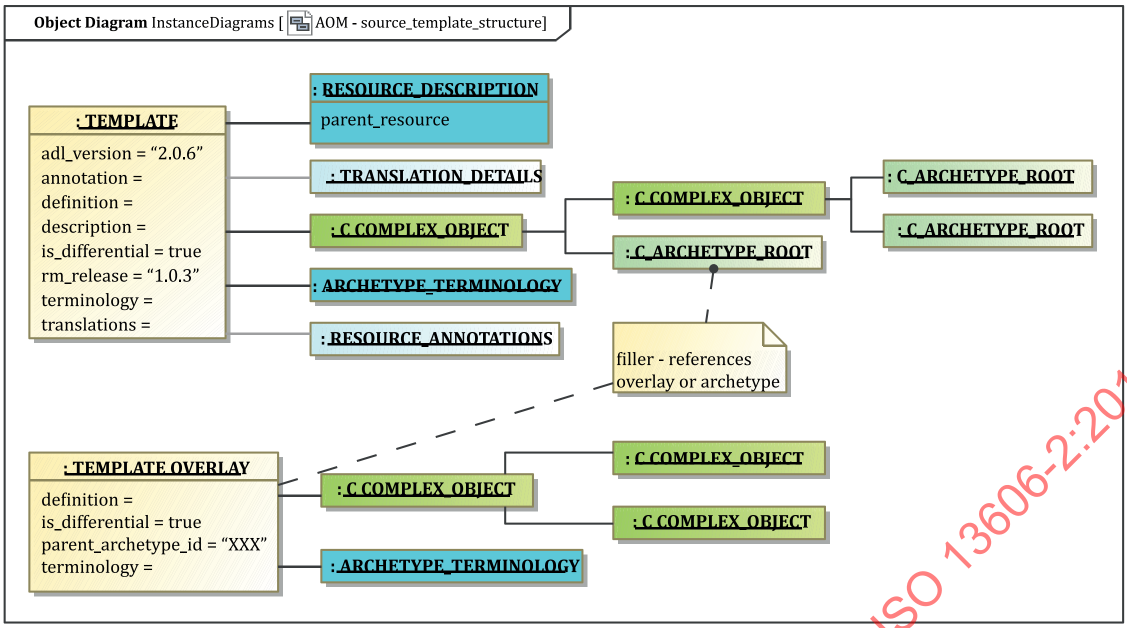


**Figure 6 — Source template structure**

Another archetype variant, also shown in Figure 6: Source template structure is the template overlay, i.e. an instance of `TEMPLATE_OVERLAY`. These are purely local components of templates, and include only the definition and terminology. The definition structure is always a specialised overlay on something else, and may not contain any slot fillers or external references, i.e. no `C_ARCHETYPE_ROOT` objects. No identifier, adl_version, languages or description are required, as they are considered to be propagated from the owning root template. Accordingly, template overlays act like a simplified specialised archetype. Template overlays can be thought of as being similar to 'anonymous' or 'inner' classes in some object-oriented programming languages.

Figure 7: Operational template structure illustrates the resulting operational template, or compiled form of a template. This is created by building the composition of referenced archetypes and/or templates and/or template overlays in their flattened form, to generate a single 'giant' archetype. The root node of this archetype, along with every archetype/template root node within, is represented using a `C_ARCHETYPE_ROOT` object. An operational template also has a component_terminologies property containing the ontologies from every constituent archetype, template and overlay.

| **Class** | ***ARCHETYPE (abstract)*** | |
|---|---|---|
| **Description** | The ARCHETYPE class defines the core formal model of the root object of any archetype or template. It includes only basic identification information, and otherwise provides the structural connections from the Archetype to its constituent parts, i.e. definition (a C_COMPLEX_OBJECT), terminology (ARCHEYTPE_TERMINOLOGY) and so on. It is the parent class of all concrete types of archetype. | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **parent_archetype_id**: `String` | Archetype reference of the specialisation parent of this archetype, if applicable. May take the form of an archetype interface identifier, i.e. the identifier up to the major version only, or can be deeper. |
| **1..1** | **archetype_id**: `ARCHETYPE_HRID` | Identifier of this archetype. |
| **1..1** | **is_differential**: `Boolean` | Flag indicating whether this archetype is differential or flat in its contents. Top-level source archetypes have this flag set to True. |
| **1..1** | **definition**: `C_COMPLEX_OBJECT` | Root node of the definition of this archetype. |

| | | |
|---|---|---|
| **1..1** | **terminology**: `ARCHETYPE_TERMINOLOGY` | The terminology of the archetype. |
| **0..1** | **rules**: `List<RULE_STATEMENT>` | Rules relating to this archetype. Statements are expressed in first order predicate logic, and usually refer to at least two attributes. |
| **Functions** | **Signature** | **Meaning** |
| | **concept_code**: `String` ***post-condition***: Result.is_equal (definition.node_id) | The concept code of the root object of the archetype, also standing for the concept of the archetype as a whole. |
| | **physical_paths**: `List<String>` | Set of language-independent paths extracted from archetype. Paths obey Xpath-like syntax and are formed from alternations of C_OBJECT.node_id and C_ATTRIBUTE.rm_attribute_name values. |
| | **logical_paths** (lang: `String`): `List<String>` | Set of language-dependent paths extracted from archetype. Paths obey the same syntax as physical_paths, but with node_ids replaced by their meanings from the ontology. |
| | **specialisation_depth**: `Integer` ***post-condition***: Result = terminology.specialisation_depth | Specialisation depth of this archetype; larger than 0 if this archetype has a parent. Derived from terminology.specialisation_depth. |
| | **is_specialised**: `Boolean` ***post-condition***: Result implies parent_archetype_hrid /= Void | True if this archetype is a specialisation of another. |
| **Invariant** | ***Invariant_concept_valid***: terminology.has_term_code (concept_code) | |
| | ***Invariant_specialisation_validity***: is_specialised implies specialisation_depth > 0 | |

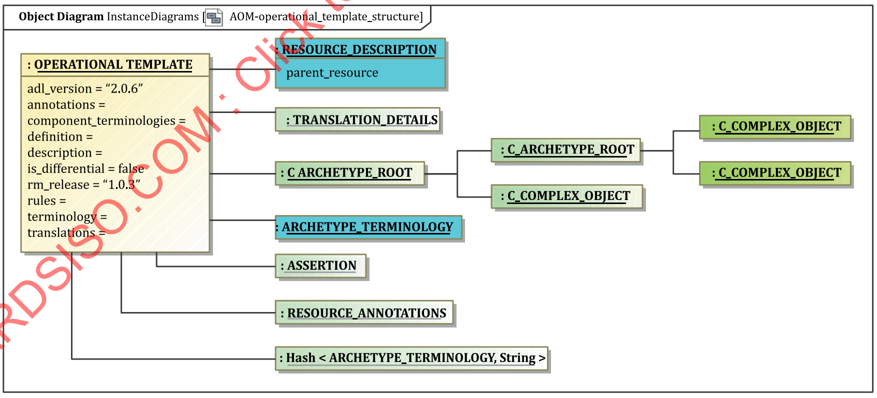


**Figure 7 — Operational template structure**

More details of template development, representation and semantics are described in the next subclause.

### 7.3.6 Class descriptions

#### 7.3.6.1 AUTHORED_RESOURCE Class

| **Class** | ***AUTHORED_RESOURCE (abstract)*** | |
|---|---|---|
| **Description** | Abstract idea of an online resource created by a human author. | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **original_language**: `TERMINOLOGY_CODE` | Language in which this resource was initially authored. Although there is no language primacy of resources overall, the language of original authoring is required to ensure natural language translations can preserve quality. Language is relevant in both the description and ontology sections. |
| **0..1** | **is_controlled**: `Boolean` | True if this resource is under any kind of change control (even file copying), in which case revision history is created. |
| **0..1** | **description**: `RESOURCE_DESCRIPTION` | Description and lifecycle information of the resource. |
| **0..1** | **uid**: `UUID` | Unique identifier of the family of archetypes having the same interface identifier (same major version). |
| **0..1** | **annotations**: `RESOURCE_ANNOTATIONS` | Annotations on individual items within the resource, keyed by path. The inner table takes the form of a Hash table of String values keyed by String tags. |
| **0..1** | **translations**: `Hash<TRANSLATION_DETAILS, String>` | List of details for each natural translation made of this resource, keyed by language. For each translation listed here, there shall be corresponding sections in all language-dependent parts of the resource. The original_language does not appear in this list. |
| **Functions** | **Signature** | **Meaning** |
| | **current_revision**: `String`<br>***Post***: Result = revision_history.most_recent_version | Most recent revision in revision_history if is_controlled else (uncontrolled) . |
| | **languages_available**: `List<String>` | Total list of languages available in this resource, derived from original_language and translations. |
| **Invariant** | ***Original_language_valid***: code_set (Code_set_id_languages).has_code (original_language.as_string) | |
| | ***Current_revision_valid***: (current_revision /= Void and not is_controlled) implies current_revision.is_equal (“(uncontrolled)”) | |
| | ***Translations_valid***: translations /= Void implies (not translations.is_empty and not translations.has (orginal_language.code_string)) | |
| | ***Description_valid***: translations /= Void implies (description.details.for_all (d \| translations.has_key (d.language.code_string))) | |
| | ***Languages_available_valid***: languages_available.has (original_language) | |
| | ***Revision_history_valid***: is_controlled xor revision_history = Void | |

### 7.3.6.2 RESOURCE_DESCRIPTION class

| Class | RESOURCE_DESCRIPTION | |
|---|---|---|
| Description | Defines the descriptive meta-data of a resource. | |
| Attributes | Signature | Meaning |
| 1..1 | **original_author**: `Hash<String, String>` | Original author of this resource, with all relevant details, including organisation. |
| 0..1 | **original_namespace**: `String` | Namespace of original author's organisation, in reverse internet form, if applicable. |
| 0..1 | **original_publisher**: `String` | Plain text name of organisation that originally published this artefact, if any. |
| 0..1 | **other_contributors**: `List<String>` | Other contributors to the resource, each listed in "name <email>" form. |
| 1..1 | **lifecycle_state**: `TERMINOLOGY_CODE` | Lifecycle state of the resource, typically including states such as: initial, in_development, in_review, published, superseded, obsolete. |
| 1..1 | **parent_resource**: `AUTHORED_RESOURCE =` | Reference to owning resource. |
| 0..1 | **custodian_namespace**: `String` | Namespace in reverse internet id form, of current custodian organisation. |
| 0..1 | **custodian_organisation**: `String` | Plain text name of current custodian organisation. |
| 0..1 | **copyright**: `String` | Optional copyright statement for the resource as a knowledge resource. |
| 0..1 | **licence**: `String` | Licence of current artefact, in format "short licence name <URL of licence>", e.g. "Apache 2.0 License <http://www.apache.org/licenses/LICENSE-2.0.html>" |
| 0..1 | **ip_acknowledgements**: `Hash<String, String>` | List of acknowledgements of other IP directly referenced in this archetype, typically terminology codes, ontology ids etc. Recommended keys are the widely known name or namespace for the IP source, as shown in the following example:<br><br>ip_acknowledgements = < ["loinc"] = <"This content from LOINC® is copyright © 1995 Regenstrief Institute, Inc. and the LOINC Committee, and available at no cost under the license at http://loinc.org/terms-of-use">; ["snomedct"] = <"Content from SNOMED CT® is copyright © 2007 IHTSDO <ihtsdo.org>">> |
| 0..1 | **references**: `Hash<String, String>` | List of references of material on which this artefact is based, as a keyed list of strings. The keys should be in a standard citation format. |
| 0..1 | **resource_package_uri**: `String` | URI of package to which this resource belongs. |

| | | |
|---|---|---|
| **0..1** | **conversion_details**: `Hash<String, String>` | Details related to conversion process that generated this model from an original, if relevant, as a list of name/value pairs. Typical example with recommended tags: conversion_details = < ["source_model"] = <"CEM model xyz ["tool"] = <"ce-m2adl v6.3.0"> ["time"] = <"2014-11-03T09:05:00">> |
| **0..1** | **other_details**: `Hash<String, String>` | Additional non language-sensitive resource meta-data, as a list of name/value pairs. |
| **0..1** | **details**: `Hash<RESOURCE_DESCRIPTION_ ITEM, String>` | Details of all parts of resource description that are natural language-dependent, keyed by language code. |

### 7.3.6.3 RESOURCE_DESCRIPTION_ITEM class

| **Class** | **RESOURCE_DESCRIPTION_ITEM** | |
|---|---|---|
| **Description** | Language-specific detail of resource description. When a resource is translated for use in another language environment, each RESOURCE_DESCRIPTION_ITEM needs to be copied and translated into the new language. | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **language**: `TERMINOLOGY_CODE` | The localised language in which the items in this description item are written. |
| **1..1** | **purpose**: `String` | Purpose of the resource. |
| **0..1** | **keywords**: `List<String>` | Keywords which characterise this resource, used e.g. for indexing and searching. |
| **0..1** | **use**: `String` | Description of the uses of the resource, i.e. contexts in which it could be used. |
| **0..1** | **misuse**: `String` | Description of any misuses of the resource, i.e. contexts in which it should not be used. |
| **0..1** | **original_resource_uri**: `List<Hash<String, String>>` | URIs of original clinical document(s) or description of which resource is a formalisation, in the language of this description item; keyed by meaning. |
| **0..1** | **other_details**: `Hash<String, String>` | Additional language-sensitive resource metadata, as a list of name/value pairs. |

### 7.3.6.4 RESOURCE_ANNOTATIONS class

| **Class** | **RESOURCE_ANNOTATIONS** |
|---|---|

| Description | Object representing annotations on an archetype. These can be of various forms, with a documentation form defined so far, which has a multi-level tabular structure [ [ [String value, String key], path key], language key]. Example instance, showing the documentation structure. |
|---|---|

```
  documentation = <

      ["en"] = <

          ["/data[id2]"] = <

              ["ui"] = <"passthrough">

>

          ["/data[id2]/items[id3]"] = <

              ["design note"] = <"this is a design note on Statement">

              ["requirements note"] = <"this is a requirements note on
Statement">

              ["medline ref"] = <"this is a medline ref on Statement">

>

>

>
```

Other sub-structures might have different keys, e.g. based on programming languages, UI toolkits etc.

| Attributes | Signature | Meaning |
|---|---|---|
| 1..1 | **documentation**: `Hash<Hash<Hash<String, String>, String>, String>` | Documentary annotations in a multi-level keyed structure. |

### 7.3.6.5 TRANSLATION_DETAILS class

| Class | TRANSLATION_DETAILS | |
|---|---|---|
| Description | Class providing details of a natural language translation. | |
| Attributes | Signature | Meaning |
| 1..1 | **language**: `TERMINOLOGY_CODE` | Language of the translation. |
| 1..1 | **author**: `Hash<String, String>` | Translator name and other demographic details. |
| 0..1 | **accreditation**: `String` | Accreditation of translator, usually a national translator's registration or association membership id. |
| 0..1 | **other_details**: `Hash<String, String>` | Any other meta-data. |
| 0..1 | **version_last_translated**: `String` | Version of this resource last time it was translated into the language represented by this TRANSLATION_DETAILS object. |

### 7.3.6.6 ARCHETYPE class

| Class | *ARCHETYPE (abstract)* | |
|---|---|---|
| Description | The ARCHETYPE class defines the core formal model of the root object of any archetype or template. It includes only basic identification information, and otherwise provides the structural connections from the Archetype to its constituent parts, i.e. definition (a C_COMPLEX_OBJECT), terminology (ARCHEYTPE_TERMINOLOGY) and so on. It is the parent class of all concrete types of archetype. | |
| Attributes | Signature | Meaning |

| | | |
|---|---|---|
| **0..1** | **parent_archetype_id**: `String` | Archetype reference of the specialisation parent of this archetype, if applicable. May take the form of an archetype interface identifier, i.e. the identifier up to the major version only, or can be deeper. |
| **1..1** | **archetype_id**: `ARCHETYPE_HRID` | Identifier of this archetype. |
| **1..1** | **is_differential**: `Boolean` | Flag indicating whether this archetype is differential or flat in its contents. Top-level source archetypes have this flag set to True. |
| **1..1** | **definition**: `C_COMPLEX_OBJECT` | Root node of the definition of this archetype. |
| **1..1** | **terminology**: `ARCHETYPE_TERMINOLOGY` | The terminology of the archetype. |
| **0..1** | **rules**: `List<RULE_STATEMENT>` | Rules relating to this archetype. Statements are expressed in first order predicate logic, and usually refer to at least two attributes. |
| **Functions** | **Signature** | **Meaning** |
| | **concept_code**: `String` <br> ***post-condition***: Result.is_equal (definition.node_id) | The concept code of the root object of the archetype, also standing for the concept of the archetype as a whole. |
| | **physical_paths**: `List<String>` | Set of language-independent paths extracted from archetype. Paths obey Xpath-like syntax and are formed from alternations of C_OBJECT.node_id and C_ATTRIBUTE.rm_attribute_name values. |
| | **logical_paths** (lang: `String`): `List<String>` | Set of language-dependent paths extracted from archetype. Paths obey the same syntax as physical_paths, but with node_ids replaced by their meanings from the ontology. |
| | **specialisation_depth**: `Integer` <br> ***post-condition***: Result = terminology.specialisation_depth | Specialisation depth of this archetype; larger than 0 if this archetype has a parent. Derived from terminology.specialisation_depth. |
| | **is_specialised**: `Boolean` <br> ***post-condition***: Result implies parent_archetype_hrid /= Void | True if this archetype is a specialisation of another. |
| **Invariant** | ***Invariant_concept_valid***: terminology.has_term_code (concept_code) | |
| | ***Invariant_specialisation_validity***: is_specialised implies specialisation_depth > 0 | |

### 7.3.6.7 AUTHORED_ARCHETYPE class

| | | |
|---|---|---|
| **Class** | **AUTHORED_ARCHETYPE** | |
| **Description** | Root object of a standalone, authored archetype, including all meta-data, description, other identifiers and lifecycle. | |
| **Inherit** | **ARCHETYPE, AUTHORED_RESOURCE** | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **adl_version**: `String` | ADL version if archetype was read in from an ADL sharable archetype. |
| **1..1** | **build_uid**: `UID` | Unique identifier of this archetype artefact instance. A new identifier is assigned every time the content is changed by a tool. Used by tools to distinguish different revisions and/or interim snapshots of the same artefact. |
| **1..1** | **rm_release**: `String` | Semver.org compatible release of the most recent reference model release on which the archetype in its current version is based. This does not imply conformance only to this release, since an archetype might be valid with respect to multiple releases of a reference model. |

| | | |
|---|---|---|
| **1..1** | **is_generated**: Boolean | If True, indicates that this artefact was machine-generated from some other source, in which case, tools would expect to overwrite this artefact on a new generation. Editing tools should set this value to False when a user starts to manually edit an archetype. |
| **1..1** | **other_meta_data**: Hash<String, String> | |
| **Invariant** | ***Invariant_adl_version_validity***: valid_version_id (adl_version) | |
| | ***Invariant_rm_release***: valid_version_id (rm_release) | |

### 7.3.6.8 ARCHETYPE_HRID class

| **Class** | **ARCHETYPE_HRID** | |
|---|---|---|
| **Description** | Human_readable structured identifier (HRID) for an archetype or template. | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **namespace**: String | Reverse domain name namespace identifier. |
| **1..1** | **rm_publisher**: String | Name of the Reference Model publisher. |
| **1..1** | **rm_package**: String | Name of the package in whose reachability graph the rm_class class is found (there can be more than one possibility in many reference models). |
| **1..1** | **rm_class**: String | Name of the root class of this archetype. |
| **1..1** | **concept_id**: String | The short concept name of the archetype as used in the multi-axial archetype_hrid. |
| **1..1** | **release_version**: String | The full numeric version of this archetype consisting of 3 parts, e.g. 1.8.2. The archetype_hrid feature includes only the major version. |
| **1..1** | **version_status**: VERSION_STATUS | The status of the version, i.e. released, release_candidate etc. |
| **1..1** | **build_count**: String | The build count since last increment of any version part. |
| **Functions** | **Signature** | **Meaning** |
| | **semantic_id**: String | The 'interface' form of the HRID, i.e. down to the major version. |
| | **physical_id**: String | The 'physical' form of the HRID, i.e. with complete version information. |
| | **version_id**: String | Full version identifier string, based on release_version and lifecycle, e.g. 1.8.2-rc.4. |
| | **major_version**: String | Major version of this archetype, extracted from release_version. |
| | **minor_version**: String | Minor version of this archetype, extracted from release_version. |
| | **patch_version**: String | Patch version of this archetype, extracted from release_version. Equivalent to patch version in patch version in semver.org system. |
| **Invariant** | ***Inv_rm_publisher_validity***: not rm_publisher.is_empty | |
| | ***Inv_rm_package_validity***: not rm_package.is_empty | |
| | ***Inv_class_name_validity***: not rm_class.is_empty | |
| | ***Inv_concept_id_validity***: not concept_id.is_empty | |
| | ***Inv_release_version_validity***: valid_version (release_version) | |

### 7.3.6.9 TEMPLATE class

| Class | TEMPLATE | |
|---|---|---|
| Description | Class representing source template, i.e. a kind of archetype that may include template overlays, and may be restricted by tools to only defining mandations, prohibitions, and restrictions on elements already defined in the flat parent. | |
| Inherit | AUTHORED_ARCHETYPE | |
| Attributes | Signature | Meaning |
| 0..1 | **overlays**: `List<TEMPLATE_OVERLAY>` | Overlay archetypes, i.e. partial archetypes that include full definition and terminology, but logically derive all their meta-data from the owning template. |
| Invariant | ***Inv_is_specialised***: is_specialised | |

### 7.3.6.10 TEMPLATE_OVERLAY class

| Class | TEMPLATE_OVERLAY |
|---|---|
| Description | A concrete form of the bare ARCHETYPE class, used to represent overlays in a source template. Overlays have no meta-data of their own, and are instead documented by their owning template. |
| Inherit | ARCHETYPE |
| Invariant | ***Inv_is_specialised***: is_specialised |

### 7.3.6.11 OPERATIONAL_TEMPLATE class

| Class | OPERATIONAL_TEMPLATE | |
|---|---|---|
| Description | Root object of an operational template. An operational template is derived from a TEMPLATE definition and the ARCHETYPEs and/or TEMPLATE_OVERLAYs mentioned by that template by a process of flattening, and potentially removal of unneeded languages and terminologies.<br><br>An operational template is used for generating and validating canonical EHR data, and also as a source artefact for generating other downstream technical artefacts, including XML schemas, APIs and UI form definitions. | |
| Inherit | AUTHORED_ARCHETYPE | |
| Attributes | Signature | Meaning |
| 0..1 | **component_terminologies**: `Hash<ARCHETYPE_TERMINOLOGY, String>` | Compendium of flattened terminologies of archetypes externally referenced from this archetype, keyed by archetype identifier. This will almost always be present in a template. |
| 0..1 | **terminology_extracts**: `Hash<ARCHETYPE_TERMINOLOGY, String>` | Directory of term definitions as a two-level table. The outer hash keys are term codes, e.g. "at4", and the inner hash key are term attribute names, e.g. "text", "description" etc. |
| Functions | Signature | Meaning |
| | **component_terminology** (an_id: `String`): `ARCHETYPE_TERMINOLOGY` | |
| Invariant | ***Specialisation_validity***: is_specialised | |

### 7.3.7 Validity rules

The following validity rules apply to all varieties of `ARCHETYPE` object.

— **VARAV**: ADL version validity. The `adl_version` top-level meta-data item, if provided, shall exist and consist of a valid 3-part version identifier.

— **VARRV**: RM release validity. The `rm_release` top-level meta-data item shall exist and consist of a valid 3-part version identifier.

— **VARCN**: archetype concept validity. The node_id of the root object of the archetype shall be of the form id1\{.1}*, where the number of '.1' components equals the specialisation depth, and shall be defined in the terminology.

— **VATDF**: value code validity. Each value code (at-code) used in a term constraint in the archetype definition shall be defined in the term_definitions part of the terminology of the flattened form of the current archetype.

— **VACDF**: constraint code validity. Each value set code (ac-code) used in a term constraint in the archetype definition shall be defined in the term_definitions part of the terminology of the current archetype.

— **VATDA**: value set assumed value code validity. Each value code (at-code) used as an assumed_value for a value set in a term constraint in the archetype definition shall exist in the value set definition in the terminology for the identified value set.

— **VETDF**: external term validity. Each external term used within the archetype definition shall exist in the relevant terminology (subject to tool accessibility; codes for inaccessible terminologies should be flagged with a warning indicating that no verification was possible).

— **VOTM**: terminology translations validity. Translations shall exist for term_definitions and constraint_definitions sections for all languages defined in the description / translations section.

— **VOKU**: object key unique. Within any keyed list in an archetype, including the description, terminology, and annotations sections, each item shall have a unique key with respect to its siblings.

— **VARDT**: archetype definition typename validity. The typename mentioned in the outer block of the archetype definition section shall match the type mentioned in the first segment of the archetype id.

— **VRANP**: annotation path valid. Each path mentioned in an annotation within the annotations section shall either be a valid archetype path, or a 'reference model' path, i.e. a path that is valid for the root class of the archetype.

— **VRRLP**: rule path valid. Each path mentioned in a rule in the rules section shall be found within the archetype, or be an RM-valid extension of a path found within the archetype.

The following validity rules apply to `ARCHETYPE` objects for which `is_overlay` = False.

— **VARID**: archetype identifier validity. The archetype shall have an identifier that conforms to the specification for archetype identifiers.

— **VDEOL**: original language specified. An `original_language` section containing the meta-data of the original authoring language shall exist.

— **VARD**: description specified. A description section containing the main meta-data of the archetype shall exist.

The following rules apply to specialised archetypes.

— **VASID**: archetype specialisation parent identifier validity. The archetype identifier stated in the specialise clause shall be the identifier of the immediate specialisation parent archetype.

— **VALC**: archetype language conformance. The languages defined in a specialised archetype shall be the same as or a subset of those defined in the flat parent.

— **VACSD**: archetype concept specialisation depth. The specialisation depth of the concept code shall be one greater than the specialisation depth of the parent archetype.

— **VATCD**: archetype code specialisation level validity. Each archetype term (‘at’ code) and constraint code (‘ac’ code) used in the archetype definition part shall have a specialisation level no greater than the specialisation level of the archetype.

## 7.4 Constraint model package

### 7.4.1 Overview

Figure 8 and Figure 9 illustrate the object model of constraints used in an archetype definition. This model is completely generic, and is designed to express the semantics of constraints on instances of classes which are themselves described in any orthodox object-oriented formalism, such as UML. Accordingly, the major abstractions in this model correspond to major abstractions in object-oriented formalisms, including several variations of the notion of 'object' and the notion of 'attribute'. The notion of 'object' rather than 'class' or 'type' is used because archetypes are about constraints on data (i.e. 'instances', or 'objects') rather than models, which are constructed from 'classes'. In this document, the word 'attribute' refers to any data property of a class, regardless of whether regarded as a 'relationship' (i.e. association, aggregation, or composition) or 'primitive' (i.e. value) attribute in an object model.

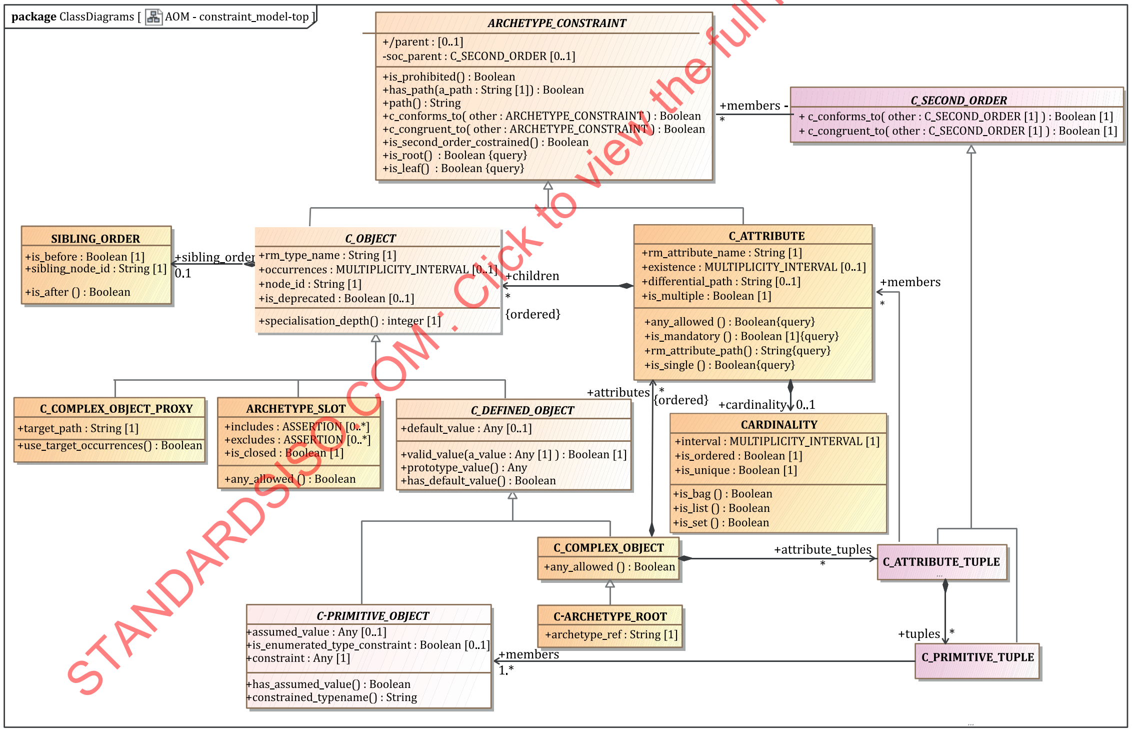

package ClassDiagrams [ AOM - constraint_model-top ]
ARCHETYPE_CONSTRAINT
+/parent : [0..1]
-soc_parent : C_SECOND_ORDER [0..1]
+is_prohibited() : Boolean
+has_path(a_path : String [1]) : Boolean
+path() : String
+c_conforms_to( other : ARCHETYPE_CONSTRAINT ) : Boolean
+c_congruent_to( other : ARCHETYPE_CONSTRAINT ) : Boolean
+is_second_order_costrained() : Boolean
+is_root() : Boolean {query}
+is_leaf() : Boolean {query}
+members -
*
C_SECOND_ORDER
+ c_conforms_to( other : C_SECOND_ORDER [1] ) : Boolean [1]
+ c_congruent_to( other : C_SECOND_ORDER [1] ) : Boolean [1]
SIBLING_ORDER
+is_before : Boolean [1]
+sibling_node_id : String [1]
+is_after () : Boolean
+sibling_order
0.1
C_OBJECT
+rm_type_name : String [1]
+occurrences : MULTIPLICITY_INTERVAL [0..1]
+node_id : String [1]
+is_deprecated : Boolean [0..1]
+specialisation_depth() : integer [1]
+children
*
{ordered}
C_ATTRIBUTE
+rm_attribute_name : String [1]
+existence : MULTIPLICITY_INTERVAL [0..1]
+differential_path : String [0..1]
+is_multiple : Boolean [1]
+any_allowed () : Boolean{query}
+is_mandatory () : Boolean [1]{query}
+rm_attribute_path() : String{query}
+is_single () : Boolean{query}
+members
*
C_COMPLEX_OBJECT_PROXY
+target_path : String [1]
+use_target_occurrences() : Boolean
ARCHETYPE_SLOT
+includes : ASSERTION [0..*]
+excludes : ASSERTION [0..*]
+is_closed : Boolean [1]
+any_allowed () : Boolean
C_DEFINED_OBJECT
+default_value : Any [0..1]
+valid_value(a_value : Any [1] ) : Boolean [1]
+prototype_value() : Any
+has_default_value() : Boolean
+attributes
*
{ordered}
+cardinality
0..1
CARDINALITY
+interval : MULTIPLICITY_INTERVAL [1]
+is_ordered : Boolean [1]
+is_unique : Boolean [1]
+is_bag () : Boolean
+is_list () : Boolean
+is_set () : Boolean
C_COMPLEX_OBJECT
+any_allowed () : Boolean
+attribute_tuples
*
C_ATTRIBUTE_TUPLE
...
C-PRIMITIVE_OBJECT
+assumed_value : Any [0..1]
+is_enumerated_type_constraint : Boolean [0..1]
+constraint : Any [1]
+has_assumed_value() : Boolean
+constrained_typename() : String
C-ARCHETYPE_ROOT
+archetype_ref : String [1]
+members
1.*
+tuples
*
C_PRIMITIVE_TUPLE

Figure 8 — constraint_model package

The definition part of an archetype is an instance of a `C_COMPLEX_OBJECT` and consists of alternate layers of object and attribute constrainer nodes, each containing the next level of nodes. At the leaves are primitive object constrainer nodes constraining primitive types such as `String`, `Integer` etc. There are also nodes that represent internal references to other nodes, constraint reference nodes that refer to a text constraint in the constraint binding part of the archetype terminology, and archetype constraint nodes, which represent constraints on other archetypes allowed to appear at a given point. The full list of concrete node types is as follows.

— `C_COMPLEX_OBJECT` : any interior node representing a constraint on instances of some non-primitive type, e.g. `OBSERVATION` , `SECTION`.

— `C_ATTRIBUTE` : a node representing a constraint on an attribute (i.e. UML 'relationship' or 'primitive attribute') in an object type.

— `C_PRIMITIVE_OBJECT` : an node representing a constraint on a primitive (built-in) object type.

— `C_COMPLEX_OBJECT_PROXY` : a node that refers to a previously defined `C_COMPLEX_OBJECT` node in the same archetype. The reference is made using a path.

— `ARCHETYPE_SLOT` : a node whose statements define a constraint that determines which other archetypes can appear at that point in the current archetype. It can be thought of like a keyhole, into which few or many keys might fit, depending on how specific its shape is. Logically it has the same semantics as a `C_COMPLEX_OBJECT` , except that the constraints are expressed in another archetype, not the current one.

— `C_ARCHETYPE_ROOT` : stands for the root node of an archetype; enables another archetype to be referenced from the present one. Used in both archetypes and templates.

The constraints define which configurations of reference model class instances are considered to conform to the archetype. For example, certain configurations of the classes `PARTY` , `ADDRESS` , `CLUSTER` and `ELEMENT` might be defined by a Person archetype as allowable structures for 'people with identity, contacts, and addresses'. Because the constraints allow optionality, cardinality and other choices, a given archetype usually corresponds to a set of similar configurations of objects.

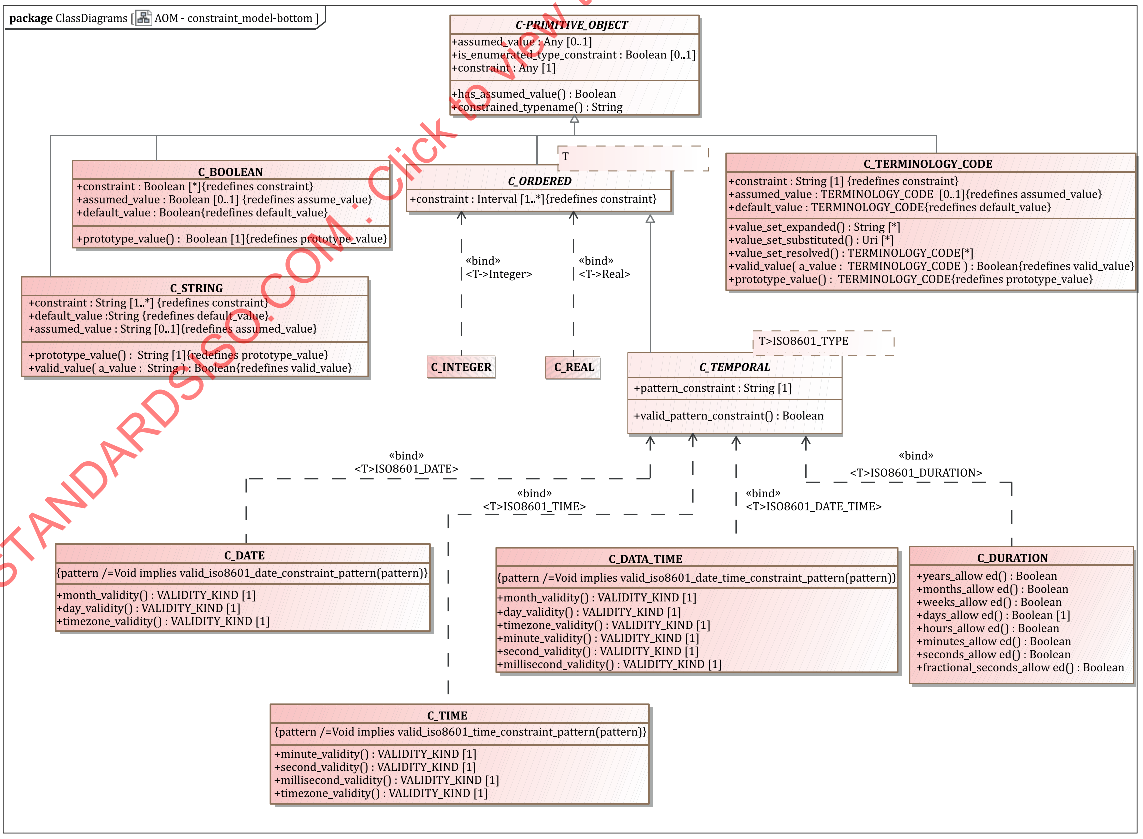

package ClassDiagrams [ AOM - constraint_model-bottom ]
C-PRIMITIVE_OBJECT
+assumed_value : Any [0..1]
+is_enumerated_type_constraint : Boolean [0..1]
+constraint : Any [1]
+has_assumed_value() : Boolean
+constrained_typename() : String
C_BOOLEAN
+constraint : Boolean [*]{redefines constraint}
+assumed_value : Boolean [0..1] {redefines assume_value}
+default_value : Boolean{redefines default_value}
+prototype_value() : Boolean [1]{redefines prototype_value}
T
C_ORDERED
+constraint : Interval [1..*]{redefines constraint}
C_TERMINOLOGY_CODE
+constraint : String [1] {redefines constraint}
+assumed_value : TERMINOLOGY_CODE [0..1]{redefines assumed_value}
+default_value : TERMINOLOGY_CODE{redefines default_value}
+value_set_expanded() : String [*]
+value_set_substituted() : Uri [*]
+value_set_resolved() : TERMINOLOGY_CODE[*]
+valid_value( a_value : TERMINOLOGY_CODE ) : Boolean{redefines valid_value}
+prototype_value() : TERMINOLOGY_CODE{redefines prototype_value}
«bind»
<T->Integer>
«bind»
<T->Real>
C_STRING
+constraint : String [1..*] {redefines constraint}
+default_value :String {redefines default_value}
+assumed_value : String [0..1]{redefines assumed_value}
+prototype_value() : String [1]{redefines prototype_value}
+valid_value( a_value : String ) : Boolean{redefines valid_value}
C_INTEGER
C_REAL
T>ISO8601_TYPE
C_TEMPORAL
+pattern_constraint : String [1]
+valid_pattern_constraint() : Boolean
«bind»
<T>ISO8601_DATE>
«bind»
<T>ISO8601_DURATION>
«bind»
<T>ISO8601_TIME>
«bind»
<T>ISO8601_DATE_TIME>
C_DATE
{pattern /=Void implies valid_iso8601_date_constraint_pattern(pattern)}
+month_validity() : VALIDITY_KIND [1]
+day_validity() : VALIDITY_KIND [1]
+timezone_validity() : VALIDITY_KIND [1]
C_DATA_TIME
{pattern /=Void implies valid_iso8601_date_time_constraint_pattern(pattern)}
+month_validity() : VALIDITY_KIND [1]
+day_validity() : VALIDITY_KIND [1]
+timezone_validity() : VALIDITY_KIND [1]
+minute_validity() : VALIDITY_KIND [1]
+second_validity() : VALIDITY_KIND [1]
+millisecond_validity() : VALIDITY_KIND [1]
C_DURATION
+years_allow ed() : Boolean
+months_allow ed() : Boolean
+weeks_allow ed() : Boolean
+days_allow ed() : Boolean [1]
+hours_allow ed() : Boolean
+minutes_allow ed() : Boolean
+seconds_allow ed() : Boolean
+fractional_seconds_allow ed() : Boolean
C_TIME
{pattern /=Void implies valid_iso8601_time_constraint_pattern(pattern)}
+minute_validity() : VALIDITY_KIND [1]
+second_validity() : VALIDITY_KIND [1]
+millisecond_validity() : VALIDITY_KIND [1]
+timezone_validity() : VALIDITY_KIND [1]

**Figure 9 — constraint_model.primitive package**

The type-name nomenclature `C_XXX` used here is intended to be read as "constraint on objects of type `XXXX` ", i.e. a `C_COMPLEX_OBJECT` is a "constraint on a complex object (defined by a complex reference model type)". These type names are used below in the formal model.

### 7.4.2 Semantics

The effect of the model is to create archetype description structures that are a hierarchical alternation of object and attribute constraints. The repeated object/attribute hierarchical structure of an archetype provides the basis for using paths to reference any node in an archetype, e.g. /attributeA[objectIdX]/attribute[objectIdY]. Archetype paths follow a syntax that is a directly convertible in and out of the W3C Xpath syntax.

#### 7.4.2.1 All node types

**Path functions**

A small number of properties are defined for all node types. The path feature computes the path to the current node from the root of the archetype, while the has_path function indicates whether a given path can be found in an archetype.

**Conformance functions**

All node types include two functions that formalise the notion of conformance of a specialised archetype to a parent archetype. Both functions take an argument which shall be a corresponding node in a parent archetype, not necessarily the immediate parent. A 'corresponding' node is one found at the same or a congruent path. A congruent path is one in which one or more at-codes have been redefined in the specialised archetype.

The `c_conforms_to` function returns True if the node on which it is called is a valid specialisation of the 'other' node. The `c_congruent_to` function returns True if the node on which it is called is the same as the other node, with the possible exception of a redefined at-code. The latter might happen due to the need to restrict the domain meaning of node to a meaning narrower than that of the same node in the parent.

**Any_allowed**

The `any_allowed` function defined on some node types indicates that any value permitted by the reference model for the attribute or type in question is allowed by the archetype; its use permits the logical idea of a completely "open" constraint to be simply expressed, avoiding the need for any further substructure.

#### 7.4.2.2 Attribute nodes

Constraints on reference model attributes, including computed attributes (represented by functions with no arguments in most programming languages), are represented by instances of `C_ATTRIBUTE` . The expressible constraints include:

— *`is_multiple`*: a flag that indicates whether the `C_ATTRIBUTE` is constraining a multiply-valued (i.e. container) RM attribute or a single-valued one;

— *`existence`*: whether the corresponding instance (defined by the *`rm_attribute_name`* attribute) shall exist;

— child objects: representing allowable values of the object value(s) of the attribute.

In the case of single-valued attributes (such as Person.date_of_birth) the children represent one or more alternative object constraints for the attribute value.

For multiply-valued attributes (such as `Person.contacts: List<Contact>`), a cardinality constraint on the container can be defined. The constraint on child objects is essentially the same except that more than one of the alternatives can co-exist in the data. Figure 10: C_ATTRIBUTE variants illustrates the two possibilities.

The appearance of both `existence` and `cardinality` constraints in `C_ATTRIBUTE` deserves some explanation, especially as the meanings of these notions are often confused in object-oriented literature. An existence constraint indicates whether an object will be found in a given attribute field, while a cardinality constraint indicates what the valid membership of a container object is. `Cardinality` is only required for container objects such as `List<T>`, `Set<T>` and so on, whereas `existence` is always possible. If both are used, the meaning is as follows: the existence constraint says whether the container object will be there (at all), while the cardinality constraint says how many items shall be in the container, and whether it acts logically as a list, set or bag. Both existence and cardinality are optional in the model, since they are only needed to override the settings from the reference model.

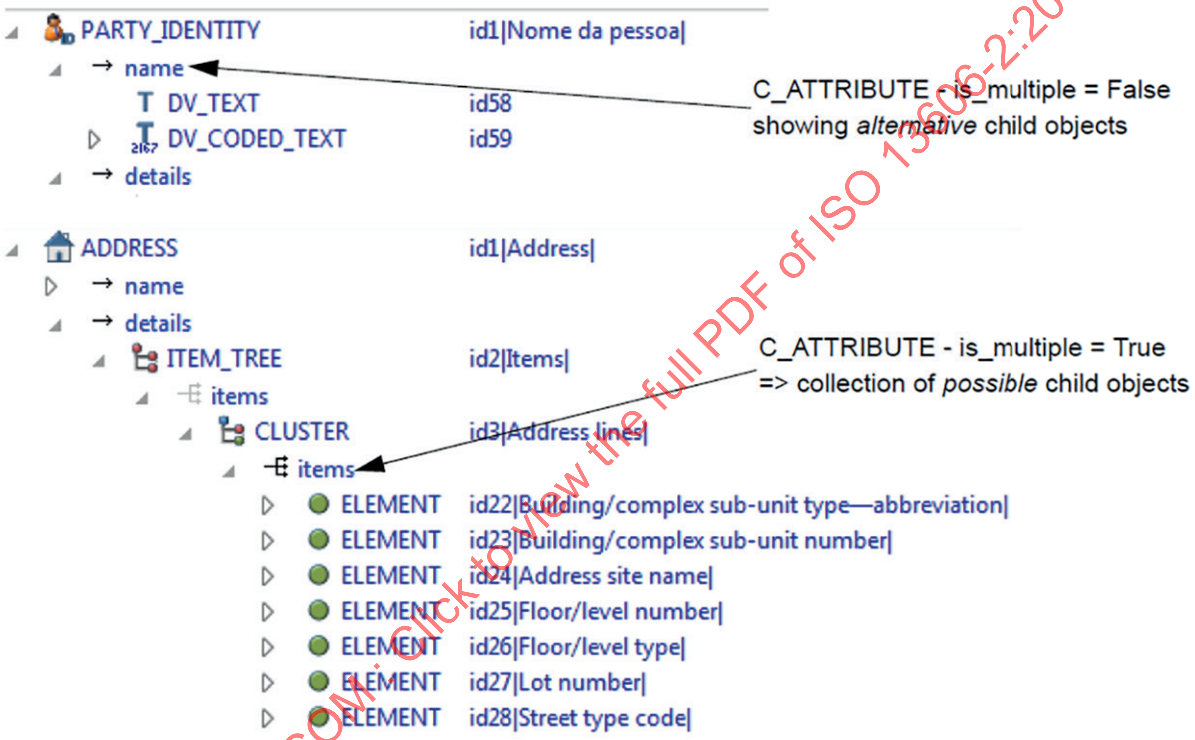


**Figure 10 — C_ATTRIBUTE variants**

### 7.4.2.3 Object node types

#### Node_id and Paths

The `node_id` attribute in the class `C_OBJECT`, inherited by all subtypes, is of key importance in the archetype constraint model. It has two functions:

— it allows archetype object constraint nodes to be individually identified, and in particular, guarantees sibling node unique identification;

— it provides a code to which a human-understanding terminology definition can be attached, as well as potentially a terminology binding.

The existence of `node_ids` in an archetype allows archetype paths to be created, which refer to each node. Every node in the archetype needs a `node_id` , but only node_ids for nodes under container attributes shall have a terminology definition. For nodes under single-valued attributes, the terminology definition is optional (and typically not supplied), since the meaning is given by the reference model attribute definition.

#### Sibling ordering

Within a specialised archetype, redefined or added object nodes may be defined under a container attribute. Since specialised archetypes are in differential form, i.e. only redefined or added nodes are expressed, not nodes inherited unchanged, the relative ordering of siblings can't be stated simply by the ordering of such items within the relevant list within the differential form of the archetype. An explicit ordering indicator is required if indeed order is specific. The `C_OBJECT.sibling_order` attribute provides this possibility. It can only be set on a `C_OBJECT` descendant within a multiply-valued attribute, i.e. an instance of `C_ATTRIBUTE` for which the `cardinality` is ordered.

**Node deprecation**

It is possible to mark an instance of any defined node type as deprecated, meaning that by preference it should not be used, and that there is an alternative solution for recording the same information. Rules or recommendations for how deprecation should be handled are outside the scope of the archetype proper, and should be provided by the governance framework under which the archetype is managed.

### 7.4.2.4 Defined object nodes (C_DEFINED_OBJECT)

The `C_DEFINED_OBJECT` subtype corresponds to the category of `C_OBJECTs` that are defined in an archetype by value, i.e. by inline definition. Four properties characterize `C_DEFINED_OBJECTs` as follows.

**Valid_value**

The `valid_value` function tests a reference model object for conformance to the archetype. It is designed for recursive implementation in which a call to the function at the top of the archetype definition would cause a cascade of calls down the tree. This function is the key function of an 'archetype-enabled kernel' component that can perform runtime data validation based on an archetype definition.

**Prototype_value**

This function is used to generate a reasonable default value of the reference object being constrained by a given node. This allows archetype-based software to build a 'prototype' object from an archetype which can serve as the initial version of the object being constrained, assuming it is being created new by user activity (e.g. via a GUI application). Implementation of this function will usually involve use of reflection libraries or similar.

**Default_value**

This attribute allows a user-specified default value to be defined within an archetype. The `default_value` object shall be of the same type as defined by the `prototype_value` function, pass the `valid_value` test. Where defined, the `prototype_value` function would return this value instead of a synthesised value.

**'Frozen' nodes**

A node may be redefined into multiple child nodes in a specialised archetype. If the children are considered to exhaustively define the value space corresponding to the original node, the latter may be 'frozen', meaning no further children can be defined. This also has a runtime implication: a frozen node cannot have any instances, only its children can.

### 7.4.2.5 Reference objects

The types `ARCHETYPE_SLOT` and `C_COMPLEX_OBJECT_PROXY` are used to express, respectively, a 'slot' where further archetypes can be used to continue describing constraints; a reference to a part of the current archetype that expresses exactly the same constraints needed at another point.

### 7.4.2.6 Complex objects (C_COMPLEX_OBJECT)

Along with `C_ATTRIBUTE`, `C_COMPLEX_OBJECT` is the key structuring type of the `constraint_model` package, and consists of attributes of type `C_ATTRIBUTE`, which are constraints on the attributes (i.e. any property, including relationships) of the reference model type. Accordingly, each `C_ATTRIBUTE` records the name of the constrained attribute (in `rm_attr_name`), the existence and cardinality expressed by the constraint (depending on whether the attribute it constrains is a multiple or single relationship), and the constraint on the object to which this C_ATTRIBUTE refers via its children attribute (according to its reference model) in the form of further C_OBJECTs .

### 7.4.2.7 Primitive types (C_PRIMITIVE_OBJECT descendants)

Constraints on primitive types are defined by the classes inheriting from C_PRIMITIVE_OBJECT , i.e. C_STRING , C_INTEGER and so on. The primitive types are represented in such a way as to accommodate both 'tuple' constraints and logically unary constraints, using a tuple array whose members are each a primitive constraint corresponding to each primitive type. Tuple constraints are second order constraints, described below, enable co-varying constraints to be stated. In the unary case, the constraint is the first member of a tuple array.

The primitive constraint for each primitive type may itself be complex. Its type is given by the type of the constraint accessor in each C_PRIMITIVE_OBJECT descendant and is summarised in the following table.

| Primitive type | Primitive constrainer type | Explanation |
|---|---|---|
| Boolean | List <Boolean> | Can represent one or two Boolean values, enabling the logical constraints 'true', 'false' and 'true or false' to be expressed. |
| String | List <String> | A list of possible string values, which may include regular expressions, which are delimited by '/' characters. |
| Terminology_code | String - [acN] or [atN]` | A string containing either a single at-code or a single ac-code. In the latter case, the constraint refers to either a locally defined value set or (via a binding) an external value set. |
| **Ordered types** | List <Interval<T>> | Can represent a single value (which is a point interval), a list of values (list of point intervals), a list of intervals, which may be mixed proper and point intervals. |
| Integer | List <Interval<Integer>> | As for Ordered type, with T = Integer |
| Real | List <Interval<Real>> | As for Ordered type, with T = Real |
| **Temporal types** | List <Interval<T→ISO8601_TYPE>><br>OR<br>String (ADL pattern) | As for ordered types, with T being an ISO8601-based type, with the addition of a second type constraint - a pattern based on ISO 8601 syntax. |
| Date | List <Interval<ISO8601_DATE>><br>OR<br>pattern | As for Temporal types with T = ISO8601_DATE |
| Time | List <Interval<ISO8601_TIME>><br>OR<br>pattern | As for Temporal types with T = ISO8601_TIME |
| Date_time | List <Interval<ISO8601_DATE_TIME>><br>OR<br>pattern | As for Temporal types with T = ISO8601_DATE_TIME |
| Duration | List <Interval<ISO8601_DURATION>><br>OR<br>pattern | As for Temporal types with T = ISO8601_DURATION |

**Assumed_value**

The `assumed_value` attribute is useful for archetypes containing any optional constraint. and provides an ability to define a value that can be assumed for a data item for which no data is found at execution time. If populated, it can contain a single at-code that shall be in the local value set referred to by the ac-code in the `constraint` attribute.

For example, an archetype for the concept 'blood pressure measurement' might contain an optional protocol section containing a data point for patient position, with choices 'lying', 'sitting' and 'standing'. Since the section is optional, data could be created according to the archetype which does not contain the protocol section. However, a blood pressure cannot be taken without the patient in some position, so clearly there is an implied value for patient position. Amongst clinicians, basic assumptions are nearly always made for such things: in general practice, the position could always safely be assumed to be "sitting" if not otherwise stated; in the hospital setting, "lying" would be the normal assumption. The assumed_value feature of archetypes allows such assumptions to be explicitly stated so that all users/systems know what value to assume when optional items are not included in the data.

Note that the notion of assumed values is distinct from that of 'default values'. The latter notion is that of a default 'pre-filled' value that is provided (normally in a local context by a template) for a data item that is to be filled in by the user, but which is typically the same in many cases. Default values are thus simply an efficiency mechanism for users. As a result, default values do appear in data, while assumed values don't.

#### 7.4.2.8 Terminology constraints (C_TERMINOLOGY_CODE)

The `C_TERMINOLOGY_CODE` type entails some complexity and merits further explanation. This is the only constrainer type whose constraint semantics are not self-contained, but located in the archetype terminology and/or in external terminologies.

A `C_TERMINOLOGY_CODE` instance in an archetype is simple: it can only be one of the following constraints:

— a single ac-code, referring to either a value-set defined in the archetype terminology or bound to an external value set or ref set;

  — in this case, an additional at-code may be included as an assumed value; the at-code shall come from the locally defined value set;

— a single at-code, representing a single possible value.

NOTE The second case in theory can be done using an ac-code referring to a value set containing a single value, but there seems little value in this extra verbiage, and little cost in providing the single-member value set short cut.

This class may be used to constrain coded types such as ISO 13606-1 CODED_VALUE or CODED_SIMPLE datatypes.

In addition, a `C_TERMINOLOGY_CODE` instance can reconstitute the internal value set via access to the archetype terminology (this has to be set up within the implementation). If bindings are evaluated, the external form of a value set can potentially be obtained as well. The utility of this is to be able to evaluate and cache certain external 'ref sets' when evaluating the Operational Template.

**Terminology code resolution**

When an archetype is deployed in the form of an operational template, the internally defined value sets, and any bindings are processed in stages in order to obtain the final terminology codes from which the user should choose. The `C_TERMINOLOGY_CODE` class provides a number of functions to formalize this as follows.

— *`value_set_expanded`*`: List<String>`: this function converts an ac-code to its corresponding set of at-codes, as defined in the `value_sets` section of the archetype.

— *`value_set_substituted`*`: List<URI>`: where bindings exist to the value set at-codes, this function converts each code to its corresponding binding target, i.e. a URI.

— *value_set_resolved*: List<TERMINOLOGY_CODE>: this function converts the list of URIs to final terms, including with textual rubrics, i.e. a list of TERMINOLOGY_CODEs.

These functions would normally be implemented as 'lambdas' or 'agents', in order to obtain access to the target terminologies.

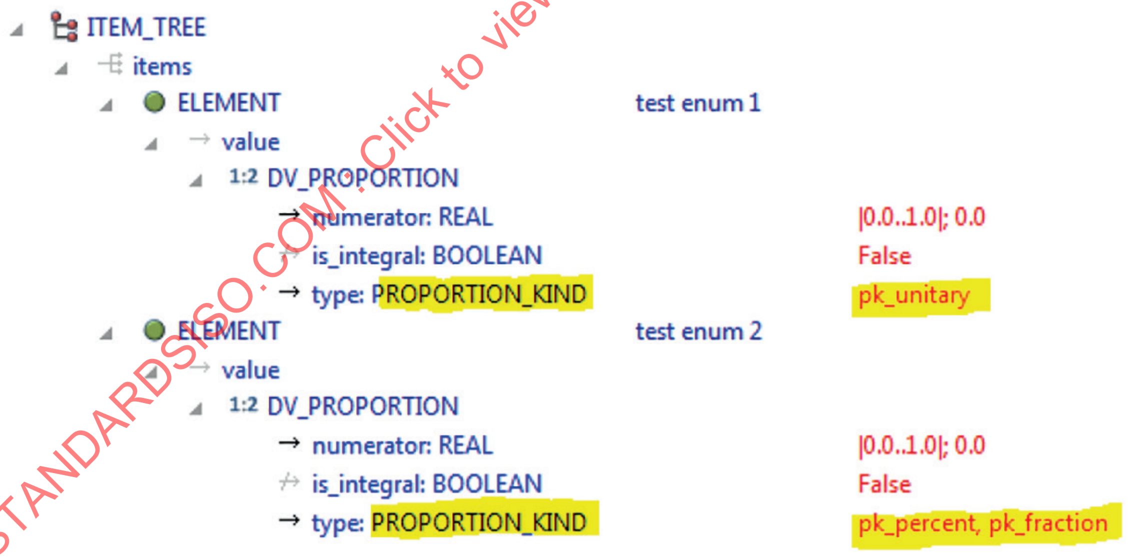
Since an archetype might not contain external terminology bindings for all (or even any) of its terminological constraints, a 'resolved' archetype will usually contain at-codes in its cADL definition. These at-codes would be treated as real coded terms in any implementation that was creating data, and as a consequence, archetype at-codes could occur in real data.

### 7.4.2.9 Constraints on enumeration types

Enumeration types in the reference model are assumed to have semantics expected in UML, and mainstream programming languages, i.e. to be a distinct type based on a primitive type, normally Integer or String. Each such type consists of a set of values from the domain of its underlying type, thus, a set of Integer, String or other primitive values. Each of these values is assumed to be named in the manner of a symbolic constant. Although strictly speaking UML does not require an enumerated type to be based on an underlying primitive type, programming languages do, hence the assumption here that values from the domain of such a type are involved.

A constraint on an enumerated type therefore consists of an AOM instance of a C_PRIMITIVE descendant, almost always C_INTEGER or C_STRING. The flag is_enumerated_type_constraint defined on C_PRIMITIVE indicates that a given C_PRIMITIVE is a constrainer for an enumerated type.

Since C_PRIMITIVEs don't have type names in ADL, the type name is inferred by any parser or compiler tool that deserialises an archetype from ADL, and stored in the rm_type attribute inherited from C_OBJECT. An example is shown below in Figure 11 of a type enumeration.



```
ITEM_TREE
  items
    ELEMENT                                   test enum 1
      value
        1:2 DV_PROPORTION
          numerator: REAL                     |0.0..1.0|; 0.0
          is_integral: BOOLEAN                False
          type: PROPORTION_KIND               pk_unitary
    ELEMENT                                   test enum 2
      value
        1:2 DV_PROPORTION
          numerator: REAL                     |0.0..1.0|; 0.0
          is_integral: BOOLEAN                False
          type: PROPORTION_KIND               pk_percent, pk_fraction
```

**Figure 11 — Enumerated constraint**

A parser that deserializes from an object dump format such as ODIN, JSON or XML will not need to do this.

The form of the constraint itself is simply a series of Integer, String or other primitive values, or an equivalent range or ranges. In the above example, the ADL equivalent of the pk_percent, pk_fraction constraint on a field of type PROPORTION_KIND is in fact just \{*2, 3*}, and it is visualised by lookup to show the relevant symbolic names.

### 7.4.3 Second order constraints

All of the constraint semantics described above can be considered 'first order' in the sense that they define how specific object/attribute/object hierarchies are defined in the instance possibility space of some part of a reference model.

Some constraints however do not fit directly within the object/attribute/object hierarchy scheme, and are considered 'second order constraints' in the archetype formalism. The 'rule' constraints constitute one such group. These constraints are defined in terms of first order predicate logic statements that can refer to any number of constraint nodes within the main hierarchy. These are described in Figure 16: Rules package.

Another type of second order constraint can be 'attached' to the object/attribute/object hierarchy in order to further limit structural possibilities. Although these constraints could also theoretically be expressed as rules, they are supported by direct additions to the main constraint model since they can be easily and intuitively represented 'inline' in ADL and corresponding AOM structures.

#### 7.4.3.1 Tuple constraints

Tuple constraints are designed to account for the very common need to constrain the values of more than one RM class attribute together. This effectively treats the attributes in question as a tuple, and the corresponding object constraints are accordingly modelled as tuples. Additions to the main constraint model to support tuples are shown in Figure 12 below.

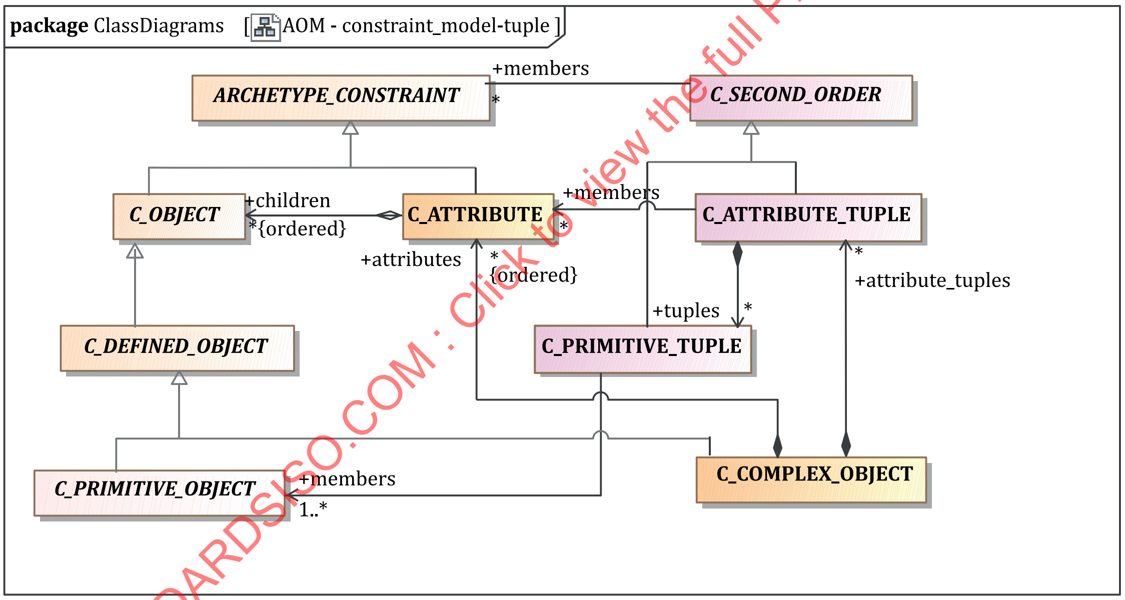


**Figure 12 — Tuple constraint model**

In this model, the type C_ATTRIBUTE_TUPLE groups the co-constrained C_ATTRIBUTE`s under a `C_COMPLEX_OBJECT. Currently the concrete type is limited to being C_PRIMITIVE_OBJECT, to reduce complexity, and since this caters for the known examples of tuple constraints. In principle, any C_DEFINED_OBJECT would be allowed, and this might change in the future.

The tuple constraint type replaces all domain-specific constraint types defined in ADL/AOM 1.4, including C_DV_QUANTITY and C_DV_ORDINAL.

These additions allow standard constraint structures (i.e. C_ATTRIBUTE / C_COMPLEX_OBJECT / C_PRIMITIVE_OBJECT hierarchies) to be 'annotated', while leaving the first order structure intact. The following example shows an archetype instance structure in which a notional ORDINAL type is constrained. The logical requirement is to constrain an ORDINAL to one of three instance possibilities, each of which consists of a pair of values for the attributes value and symbol, of type Integer and TERMINOLOGY_CODE respectively. Each of these three instance constraints should be understood as an alternative for the single valued owning attribute, ELEMENT .value. Tuple constraints achieve the requirement to express the constraints as pairs not just as allowable alternatives at the final leaf level, which would incorrectly allowing any mixing of the Integer and code values, as illustrated in Figures 13 and 14.

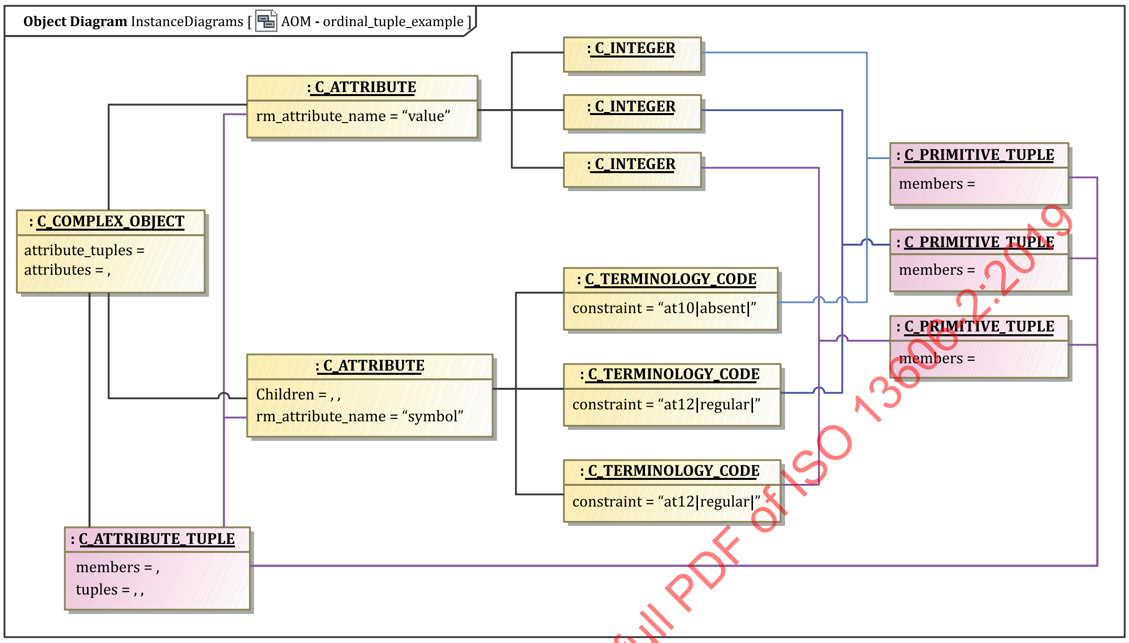
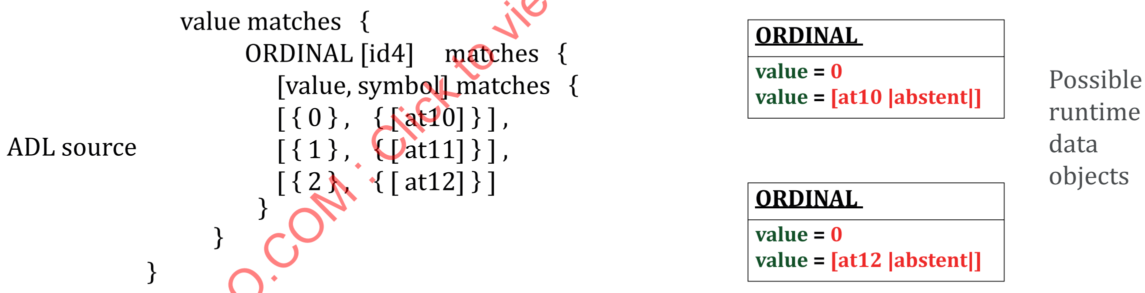


Figure 13 — Tuple constraint example AOM instances



```
value matches {
    ORDINAL [id4]    matches {
        [value, symbol] matches {
            [ { 0 } ,   { [ at10] } ] ,
            [ { 1 } ,   { [ at11] } ] ,
            [ { 2 } ,   { [ at12] } ]
        }
    }
}
```

ORDINAL
value = 0
value = [at10 |abstent|]

ORDINAL
value = 0
value = [at12 |abstent|]

Figure 14 — Tuple constraint example data

### 7.4.3.2 Assertions

Assertions are also used in ARCHETYPE_SLOTs, in order to express the 'included' and 'excluded' archetypes for the slot. In this case, each assertion is an expression that refers to parts of other archetypes, such as its identifier (e.g. 'include archetypes with short_concept_name matching xxxx '). Assertions are modelled here as a generic expression tree of unary prefix and binary infix operators.

## 7.4.4 AOM type substitutions

Specialised archetypes can redefine the types of the AOM objects from parent archetypes. Not all type substitutions are valid, so this section provides the rules for these substitutions.

The C_OBJECT types defined in Figure 8: constraint_model Package are reproduced below in Figure 15, with concrete types that may actually occur in archetypes shown in dark yellow / non-italic.

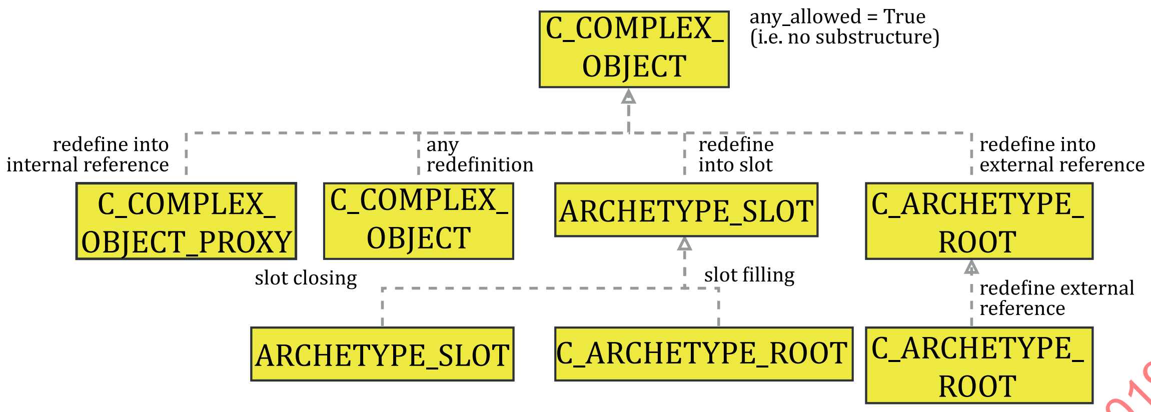


**Figure 15 — C_Object substitutions**

Within a specialised archetype, nodes that redefine corresponding nodes in the parent are normally of the same `C_OBJECT` type (we can think of this as a 'meta-type', since the RM type is the 'type' in the information model sense), but in some cases might also be of different `C_OBJECT` types.

The rules for meta-type redefinition are as follows:

— A node of each meta-type can be redefined by a node of the same meta-type, with narrowed / added constraints;

— `ARCHETYPE_SLOT` can be redefined by:

  — one or more `C_ARCHETYPE_ROOT` nodes taken together, considered to define a 'filled' version of the slot;

  — an `ARCHETYPE_SLOT` , in order to close the slot.

— A `C_ARCHETYPE_ROOT` node can be redefined by:

  — A `C_ARCHETYPE_ROOT`, where the archetype_ref of the redefining node is a specialisation of that mentioned in the parent node.

— A terminal `C_COMPLEX_OBJECT` node containing no constraint other than RM type, *`node_id`* and possibly occurrences (i.e. having no substructure), can be redefined by a constraint of any other AOM type.

The 'terminal `C_COMPLEX_OBJECT` ' can be understood as a placeholder node primarily defined for the purpose of stating meaning.

### 7.4.5 Class definitions

#### 7.4.5.1 ARCHETYPE_CONSTRAINT class

| Class | *ARCHETYPE_CONSTRAINT (abstract)* | |
|---|---|---|
| **Description** | Defines common constraints for any class in the reference model that has an archetype_id property. | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **parent:** | |
| **0..1** | **soc_parent**: `C_SECOND_ORDER` | |
| **Functions** | **Signature** | **Meaning** |
| | **is_prohibited**: `Boolean` | True if this node (and all its sub-nodes) is a valid archetype node for its type. This function should be implemented by each subtype to perform semantic validation of itself, and then call the is_valid function in any subparts, and generate the result appropriately. |
| | **has_path** (a_path: `String`): `boolean` | True if the relative path a_path exists at this node. |
| | **path**: `String` | Path of this node relative to root of archetype. |
| | **c_conforms_to** (other: `ARCHETYPE_CONSTRAINT`): `Boolean` | True if constraints represented by this node, ignoring any sub-parts, are narrower or the same as other. Typically used during validation of specialised archetype nodes. |
| | **c_congruent_to** (other: `ARCHETYPE_CONSTRAINT`): `Boolean` | True if constraints represented by this node contain no further redefinitions with respect to the node other, with the exception of node_id redefinition in C_OBJECT nodes. Typically used to test if an inherited node locally contains any constraints. |
| | **is_second_order_constrained**: `Boolean` | |
| | **is_root**: `Boolean` | |
| | **is_leaf**: `Boolean` | |

#### 7.4.5.2 C_ATTRIBUTE class

| Class | C_ATTRIBUTE | |
|---|---|---|
| **Description** | Abstract model of constraint on any kind of attribute in a class model. | |
| **Inherit** | ARCHETYPE_CONSTRAINT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **rm_attribute_name**: `String` | Reference model attribute within the enclosing type represented by a C_OBJECT. |
| **0..1** | **existence**: `MULTIPLICITY_INTERVAL` | Constraint settable on every attribute, regardless of whether it is singular or of a container type, which indicates whether its target object exists or not (i.e. is mandatory or not). Only set if it overrides the underlying reference model or parent archetype in the case of specialised archetypes. |

| | | |
|---|---|---|
| **0..1** | **children**: `List<C_OBJECT>` | Child C_OBJECT nodes. Each such node represents a constraint on the type of this attribute in its reference model. Multiples occur both for multiple items in the case of container attributes, and alternatives in the case of singular attributes. |
| **0..1** | **differential_path**: `String` | Path to the parent object of this attribute (i.e. doesn't include the name of this attribute). Used only for attributes in differential form, specialised archetypes. Enables only the re-defined parts of a specialised archetype to be expressed, at the path where they occur. |
| **0..1** | **cardinality**: `CARDINALITY` | Cardinality constraint of attribute, if a container attribute. |
| **1..1** | **is_multiple**: `Boolean` | Flag indicating whether this attribute constraint is on a container (i.e. multiply-valued) attribute. |
| **Functions** | **Signature** | **Meaning** |
| | **any_allowed**: `Boolean` | |
| | **is_mandatory**: `Boolean` | |
| | **rm_attribute_path**: `String` | Path of this attribute with respect to owning C_OBJECT, including differential path where applicable. |
| | **is_single**: `Boolean` | True if this node logically represents a single-valued attribute. Evaluated as not is_multiple. |
| **(effected)** | **c_congruent_to** (other: `ARCHETYPE_CONSTRAINT`): `Boolean`<br>***Post***: Result = existence = Void and is_single and other.is_single) or (is_multiple and other.is_multiple and cardinality = Void | True if constraints represented by this node contain no further redefinitions with respect to the node other, with the exception of node_id redefnition in C_OBJECT nodes. Typically used to test if an inherited node locally contains any constraints. |
| **(effected)** | **c_conforms_to** (other: `ARCHETYPE_CONSTRAINT`): `Boolean`<br>***Post***: Result = existence_conforms_to (other) and is_single and other.is_single) or else (is_multiple and cardinality_conforms_to (other) | True if constraints represented by this node, ignoring any sub-parts, are narrower or the same as other. Typically used during validation of specialised archetype nodes. |

**Conformance semantics**

The following functions formally define the conformance of an attribute node in a specialised archetype to the corresponding node in a parent archetype, where 'corresponding' means a node found at the same or a congruent path.

```
c_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := existence_conforms_to (other) and
            ((is_single and other.is_single) or else
            (is_multiple and cardinality_conforms_to (other)))
    end
c_congruent_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := existence = Void and ((is_single and other.is_single) or
                (is_multiple and other.is_multiple and cardinality = Void))
    end
existence_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
    do
        if existence /= Void and other.existence /= Void then
            Result := other.existence.contains (existence)
        else
            Result := True
        end
    end
cardinality_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
    do
        if cardinality /= Void and other.cardinality /= Void then
            Result := other.cardinality.contains (cardinality)
        else
            Result := True
        end
    end
```

**Validity rules**

The validity rules are as follows.

— **VCARM**: attribute name reference model validity: an attribute name introducing an attribute constraint block shall be defined in the underlying information model as an attribute (stored or computed) of the type which introduces the enclosing object block.

— **VCAEX**: archetype attribute reference model existence conformance: the existence of an attribute, if set, shall conform, i.e. be the same or narrower, to the existence of the corresponding attribute in the underlying information model.

— **VCAM**: archetype attribute reference model multiplicity conformance: the multiplicity, i.e. whether an attribute is multiply- or single-valued, of an attribute shall conform to that of the corresponding attribute in the underlying information model.

— **VDIFV**: archetype attribute differential path validity: an archetype may only have a differential path if it is specialised.

The following validity rule applies to redefinition in a specialised archetype.

— **VDIFP**: specialised archetype attribute differential path validity: if an attribute constraint has a differential path, the path shall exist in the flat parent, and also be valid with respect to the reference model, i.e. in the sense that it corresponds to a legal potential construction of objects.

— **VSANCE**: specialised archetype attribute node existence conformance: the existence of a redefined attribute node in a specialised archetype, if stated, shall conform to the existence of the corresponding node in the flat parent archetype, by having an identical range, or a range wholly contained by the latter.

— **VSAM**: specialised archetype attribute multiplicity conformance: the multiplicity, i.e. whether an attribute is multiply- or single-valued, of a redefined attribute shall conform to that of the corresponding attribute in the parent archetype.

The following validity rules apply to single-valued attributes, i.e. when `C_ATTRIBUTE.is_multiple` is False.

— **VACSO**: single-valued attribute child object occurrences validity: the occurrences of a child object of a single-valued attribute cannot have an upper limit greater than 1.

The following validity rules apply to container attributes, i.e. when C_ATTRIBUTE.is_multiple is True.

— **VACMCU**: cardinality/occurrences upper bound validity: where a cardinality with a finite upper bound is stated on an attribute, for all immediate child objects for which an occurrences constraint is stated, the occurrences shall either have an open upper bound (i.e. n..*) which is interpreted as the maximum value allowed within the cardinality, or else a finite upper bound which is ⇐ the cardinality upper bound.

— **VACMCO**: cardinality/occurrences orphans: it shall be possible for at least one instance of one optional child object (i.e. an object for which the occurrences lower bound is 0) and one instance of every mandatory child object (i.e. object constraints for which the occurrences lower bound is >= 1) to be included within the cardinality range.

— **VCACA**: archetype attribute reference model cardinality conformance: the cardinality of an attribute shall conform, i.e. be the same or narrower, to the cardinality of the corresponding attribute in the underlying information model.

The following validity warnings apply to container attributes, i.e. when `C_ATTRIBUTE.is_multiple` is True.

— **WACMCL**: cardinality/occurrences lower bound validity: where a cardinality with a finite upper bound is stated on an attribute, for all immediate child objects for which an occurrences constraint is stated, the sum of occurrences lower bounds should be lower than the cardinality upper limit.

The following validity rule applies to cardinality redefinition in a specialised archetype.

— **VSANCC**: specialised archetype attribute node cardinality conformance: the cardinality of a redefined (multiply-valued) attribute node in a specialised archetype, if stated, shall conform to the cardinality of the corresponding node in the flat parent archetype by either being identical, or being wholly contained by the latter.

### 7.4.5.3 CARDINALITY class

| **Class** | **CARDINALITY** | |
|---|---|---|
| **Description** | Express constraints on the cardinality of container objects which are the values of multiply-valued attributes, including uniqueness and ordering, providing the means to state that a container acts like a logical list, set or bag. The cardinality cannot contradict the cardinality of the corresponding attribute within the relevant reference model. | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **interval**: `MULTIPLICITY_INTERVAL` | The interval of this cardinality. |
| **1..1** | **is_ordered**: `Boolean` | True if the members of the container attribute to which this cardinality refers are ordered. |
| **1..1** | **is_unique**: `Boolean` | True if the members of the container attribute to which this cardinality refers are unique. |
| **Functions** | **Signature** | **Meaning** |
| | **is_bag**: `Boolean` | True if the semantics of this cardinality represent a bag, i.e. unordered, non-unique membership. |
| | **is_list**: `Boolean` | True if the semantics of this cardinality represent a list, i.e. ordered, non-unique membership. |
| | **is_set**: `Boolean` | True if the semantics of this cardinality represent a bag, i.e. unordered, non-unique membership. |

### 7.4.5.4 C_OBJECT class

| **Class** | ***C_OBJECT (abstract)*** | |
|---|---|---|
| **Description** | Abstract model of constraint on any kind of object node. | |
| **Inherit** | ARCHETYPE_CONSTRAINT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **rm_type_name**: `String` | Reference model type that this node corresponds to. |

| | | |
|---|---|---|
| **0..1** | **occurrences**: `MULTIPLICITY_INTERVAL` | Occurrences of this object node in the data, under the owning attribute. Upper limit can only be greater than 1 if owning attribute has a cardinality of more than 1. Only set if it overrides the parent archetype in the case of specialised archetypes, or else the occurrences inferred from the underlying reference model existence and/or cardinality of the containing attribute. |
| **1..1** | **node_id**: `String` | Semantic identifier of this node, used to distinguish sibling nodes. All nodes shall have a node_id; for nodes under a container C_ATTRIBUTE, the id shall be an id-code shall be defined in the archetype terminology. For valid structures, all node ids are id-codes. For C_PRIMITIVE_OBJECTs, it will have the special value Primitive_node_id. |
| **0..1** | **is_deprecated**: `Boolean` | True if this node and by implication all sub-nodes are deprecated for use. |
| **0..1** | **sibling_order**: `SIBLING_ORDER` | Optional indicator of order of this node with respect to another sibling. Only meaningful in a specialised archetype for a C_OBJECT within a C_ATTRIBUTE with is_multiple = True. |
| **Functions** | **Signature** | **Meaning** |
| | **specialisation_depth**: `Integer` | Level of specialisation of this archetype node, based on its node_id. The value 0 corresponds to non-specialised, 1 to first-level specialisation and so on. The level is the same as the number of '.' characters in the node_id code. If node_id is not set, the return value is -1, signifying that the specialisation level should be determined from the nearest parent C_OBJECT node having a node_id. |

**Occurrences inferencing rules**

The notion of 'occurrences' does not exist in an object model that might be used as the reference model on which archetypes are based, because it is a class model. However, archetypes make statements about how many objects conforming to a specific object constraint node might exist, within a container attribute. In an operational template, an occurrences constraint is required on all children of container attributes. Most such constraints come from the source template(s) and archetypes, but in some cases, there will be nodes with no occurrences. In these cases, the occurrences constraint is inferred from the reference model according to the following algorithm, where `c_object` represents any object node in an archetype.

```
if not c_object.is_root and c_object.occurrences = Void then
    if is_container_attribute_in_rm (c_object.parent) then
        if rm_parent_attr.cardinality.upper_unbounded then
            c_object.set_occurrences (|0..*|)
        else
            c_object.set_occurrences (|0..rm_parent_attr.cardinality.upper|)
        end
    else
        c_object.set_occurrences (rm_parent_attr.existence)
    end
end
```

Occurrences is not really required on children of single-valued attributes, because the notional occurrences is always the same as the existence constraint of the owning attribute in the flat parent structure, or else the reference model.

**Conformance semantics**

The following functions formally define the conformance of an object node in a specialised archetype to the corresponding node in a parent archetype, where 'corresponding' means a node found at the same or a congruent path.

```
c_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := node_id_conforms_to (other) and
            occurrences_conforms_to (other) and
            (rm_type_name.is_equal (other.rm_type_name) or else
            rm_types_conformant(rm_type_name, other.rm_type_name))
    end
c_congruent_to (other: like Current): Boolean
        -- True if this node makes no changes to 'other' (from a
        -- specialisation parent archetype) apart from possible
        -- change of node-id
    require
        other /= Void
    do
        Result := rm_type_name.is_case_insensitive_equal (other.rm_type_name) and
            (occurrences = Void or else occurrences ~ other.occurrences) and
            (sibling_order = Void or else sibling_order ~ other.sibling_order) and
            node_reuse_congruent (other)
    end
rm_type_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := rm_type_name.is_equal (other.rm_type_name) or
            rm_checker.is_sub_type_of (rm_type_name, other.rm_type_name)
    end
occurrences_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
        other_is_flat: other.occurrences /= Void
    do
        if occurrences /= Void and other.occurrences /= Void then
            Result := other.occurrences.contains (occurrences)
        else
            Result := True
        end
    end
node_id_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
    do
        Result := codes_conformant (node_id, other.node_id)
    end
```

## Validity rules

The validity rules for all `C_OBJECTs` are as follows.

— **VCORM** object constraint type name existence: a type name introducing an object constraint block shall be defined in the underlying information model.

— **VCORMT** object constraint type validity: a type name introducing an object constraint block shall be the same as or conform to the type stated in the underlying information model of its owning attribute.

— **VCOCD** object constraint definition validity: an object constraint block consists of one of the following (depending on subtype): an 'any' constraint; a reference; an inline definition of sub-constraints, or nothing, in the case where occurrences is set to {0}.

— **VCOID** object node identifier validity: every object node shall have a node identifier.

— **VCOSU** object node identifier validity: every object node shall be unique within the archetype.

The following validity rules govern `C_OBJECTs` in specialised archetypes.

- **VSONT** specialised archetype object node meta-type conformance: the meta-type of a redefined object node (i.e. the AOM node type such as `C_COMPLEX_OBJECT` etc.) in a specialised archetype shall be the same as that of the corresponding node in the flat parent, with the following exceptions: a `C_COMPLEX_OBJECT` with no child attributes may be redefined by a node of any AOM type; a `C_COMPLEX_OBJECT_PROXY`, may be redefined by a `C_COMPLEX_OBJECT`; a `ARCHTEYPE_SLOT` may be redefined by `C_ARCHETYPE_ROOT` (i.e. 'slot-filling'). See also validity rules VDSSID and VARXID.
- **VSONCT** specialised archetype object node reference type conformance: the reference model type of a redefined object node in a specialised archetype shall conform to the reference model type in the corresponding node in the flat parent archetype by either being identical, or conforming via an inheritance relationship in the relevant reference model.
- **VSONIN** specialised archetype new object node identifier validity: if an object node in a specialised archetype is a new node with respect to the flat parent, and it carries a node identifier, the identifier shall be a 'new' node identifier, specalised at the level of the child archetype.
- **VSONIF** specialised archetype object node identifier validity in flat siblings: the identification (or not) of an object node in a specialised archetype shall be valid with respect to any sibling object nodes in the flattened parent (see VACMI).
- **VSONCO** specialised archetype redefine object node occurrences validity: the occurrences of a redefined object node in a specialised archetype, if stated, shall conform to the occurrences in the corresponding node in the flat parent archetype by either being identical, or being wholly contained by the latter.
- **VSONPT** specialised archetype prohibited object node AOM type validity: the occurrences of a redefined object node in a specialised archetype, may only be prohibited (i.e. {0}) if the matching node in the parent is of the same AOM type.
- **VSONPI** specialised archetype prohibited object node AOM node id validity: a redefined object node in a specialised archetype with occurrences matching {0} shall have exactly the same node id as the node in the flat parent being redefined.
- **VSONPO** specialised archetype object node prohibited occurrences validity: the occurrences of a new (i.e. having no corresponding node in the parent flat) object node in a specialised archetype, if stated, may not be 'prohibited', i.e. {0}, since prohibition only makes sense for an existing node.
- **VSSM** specialised archetype sibling order validity: the sibling order node id code used in a sibling marker in a specialised archetype shall refer to a node found within the same container in the flat parent archetype.

### 7.4.5.5 SIBLING_ORDER class

| **Class** | **SIBLING_ORDER** | |
|---|---|---|
| **Description** | Defines the order indicator that can be used on a C_OBJECT within a container attribute in a specialised archetype to indicate its order with respect to a sibling defined in a higher specialisation level.<br>Misuse: This type cannot be used on a C_OBJECT other than one within a container attribute in a specialised archetype. | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **is_before**: `Boolean` | True if the order relationship is 'before', if False, it is 'after'. |
| **1..1** | **sibling_node_id**: `String` | Node identifier of sibling before or after which this node should come. |
| **Functions** | **Signature** | **Meaning** |

| | | |
|---|---|---|
| | **is_after**: Boolean | True if the order relationship is 'after', computed as the negation of is_before. |

### 7.4.5.6 C_DEFINED_OBJECT class

| Class | *C_DEFINED_OBJECT (abstract)* | |
|---|---|---|
| **Description** | Abstract parent type of C_OBJECT subtypes that are defined by value, i.e. whose definitions are actually in the archetype rather than being by reference. | |
| **Inherit** | C_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **is_frozen**: Boolean | True if this node is closed for further re-definition. Any child nodes defined as sib-lings are considered to exhaustively represent the possible value space of this original parent node. |
| **0..1** | **default_value**: Any | Default value set in a template, and present in an operational template. Generally limited to leaf and near-leaf nodes. |
| **Functions** | **Signature** | **Meaning** |
| | **valid_value** (a_value: Any): Boolean | True if a_value is valid with respect to constraint expressed in concrete instance of this type. |
| | **prototype_value**: Any | Generate a prototype value from this constraint object. |
| | **has_default_value**: Boolean | True if there is an assumed value. |

### 7.4.5.7 C_COMPLEX_OBJECT class

| Class | **C_COMPLEX_OBJECT** | |
|---|---|---|
| **Description** | Constraint on complex objects, i.e. any object that consists of other object constraints. | |
| **Inherit** | C_DEFINED_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **attributes**: List<C_ATTRIBUTE> | List of constraints on attributes of the reference model type represented by this object. |
| **0..1** | **attribute_tuples**: List<C_ATTRIBUTE_TUPLE> | List of attribute tuple constraints under this object constraint, if any. |
| **Functions** | **Signature** | **Meaning** |
| | **any_allowed**: Boolean | True if any value (i.e. instance) of the reference model type would be allowed. Redefined in descendants. |

**Validity Rules**

The validity rules for `C_COMPLEX_OBJECTs` are as follows.

— **VCATU** attribute uniqueness: sibling attributes occurring within an object node shall be uniquely named with respect to each other, in the same way as for class definitions in an object reference model.

### 7.4.5.8 C_ARCHETYPE_ROOT class

| Class | **C_ARCHETYPE_ROOT** |
|---|---|

| | | |
|---|---|---|
| **Description** | A specialisation of C_COMPLEX_OBJECT whose node_id attribute is an archetype identifier rather than the normal internal node code (i.e. id-code). Used in two situations. The first is to represent an 'external reference' to an archetype from within another archetype or template. This supports re-use. The second use is within a template, where it is used as a slot-filler.<br><br>For a new external reference, the node_id is set in the normal way, i.e. with a new code at the specialisation level of the archetype. For a slot-filler or a redefined external reference, the node_id is set to a specialised version of the node_id of the node being specialised, allowing matching to occur during flattening.<br><br>In all uses within source archetypes and templates, the children attribute is Void.<br><br>In an operational template, the node_id is converted to the archetype_ref, and the structure contains the result of flattening any template overlay structure and the underlying flat archetype. | |
| **Inherit** | C_COMPLEX_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **archetype_ref**: `String` | Reference to archetype is being used to fill a slot or redefine an external reference. Typically an 'interface' archetype id, i.e. identifier with partial version information. |

**Validity rules**

The following validity rules apply to `C_ARCHETYPE_ROOT` objects.

— **VARXS** external reference conforms to slot: the archetype reference shall conform to the archetype slot constraint of the flat parent and be of a reference model type from the same reference model as the current archetype.

— **VARXNC** external reference node identifier validity: if the reference object is a redefinition of either a slot node, or another external reference node, the node_id of the object shall conform to (i.e. be the same or a child of) the node_id of the corresponding parent node.

— **VARXAV** external reference node archetype reference validity: if the reference object is a redefinition of another external reference node, the archetype_ref of the object shall match a real archetype that has as an ancestor the archetype matched by the archetype reference mentioned in the corresponding parent node.

— **VARXTV** external reference type validity: the reference model type of the reference object archetype identifier shall be identical, or conform to the type of the slot, if there is one, in the parent archetype, or else to the reference model type of the attribute in the flat parent under which the reference object appears in the child archetype.

— **VARXR** external reference refers to resolvable artefact: the archetype reference shall refer to an artefact that can be found in the current repository.

The following validity rules apply to a `C_ARCHETYPE_ROOT` that specialises a `ARCHETYPE_SLOT` in the parent archetype:

— **VARXID** external reference slot filling id validity: an external reference node defined as a filler for a slot in the parent archetype shall have a node id that is a specialisation of that of the slot.

#### 7.4.5.9 ARCHETYPE_SLOT class

| **Class** | **ARCHETYPE_SLOT** | |
|---|---|---|
| **Description** | Constraint describing a slot' where another archetype can occur. | |
| **Inherit** | C_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |

| | | |
|---|---|---|
| **0..1** | **includes**: `List<ASSERTION>` | List of constraints defining other archetypes that could be included at this point. |
| **0..1** | **excludes**: `List<ASSERTION>` | List of constraints defining other archetypes that cannot be included at this point. |
| **1..1** | **is_closed**: `Boolean` | True if this slot specification in this artefact is closed to further filling either in further specialisations or at runtime. Default value False, i.e. unless explicitly set, a slot remains open. |
| **Functions** | **Signature** | **Meaning** |
| | **any_allowed**: `Boolean` | True if no constraints stated, and slot is not closed. |

**Validity rules**

The validity rules for `ARCHETYPE_SLOTs` are as follows.

— **VDFAI** archetype identifier validity in definition. Any archetype identifier mentioned in an archetype slot in the definition section shall conform to the specification for archetype identifiers in this document.

— **VDSIV** archetype slot 'include' constraint validity. The 'include' constraint in an archetype slot shall conform to the slot constraint validity rules.

— **VDSEV** archetype slot 'exclude' constraint validity. The 'exclude' constraint in an archetype slot shall conform to the slot constraint validity rules.

The slot constraint validity rules are as follows.

```
if includes not empty and = any then
    not (excludes empty or /= any) ==> VDSEV Error
elseif includes not empty and /= any then
    not (excludes empty or = any) ==> VDSEV Error
elseif excludes not empty and = any then
    not (includes empty or /= any) ==> VDSIV Error
elseif excludes not empty and /= any then
    not (includes empty or = any) ==> VDSIV Error
end
```

The following validity rules apply to `ARCHETYPE_SLOTs` defined as the specialisation of a slot in the parent archetype.

— **VDSSID** slot redefinition child node id: a slot node in a specialised archetype that redefines a slot node in the flat parent shall have an identical node id.

— **VDSSM** specialised archetype slot definition match validity. The set of archetypes matched from a library of archetypes by a specialised archetype slot definition shall be a proper subset of the set matched from the same library by the parent slot definition.

— **VDSSP** specialised archetype slot definition parent validity. The flat parent of the specialisation of an archetype slot shall be not be closed (is_closed = False).

— **VDSSC** specialised archetype slot definition closed validity. In the specialisation of an archetype slot, either the slot can be specified to be closed (is_closed = True) or the slot can be narrowed, but not both.

### 7.4.5.10 C_COMPLEX_OBJECT_PROXY class

| | |
|---|---|
| **Class** | **C_COMPLEX_OBJECT_PROXY** |

| | | |
|---|---|---|
| **Description** | A constraint defined by proxy, using a reference to an object constraint defined elsewhere in the same archetype. Note that since this object refers to another node, there are two objects with available occurrences values. The local occurrences value on a COJMPLEX_OBJECT_PROXY should always be used; when setting this from a serialised form, if no occurrences is mentioned, the target occurrences should be used (not the standard default of {1..1}); otherwise the locally specified occurrences should be used as normal. When serialising out, if the occurrences is the same as that of the target, it can be left out. | |
| **Inherit** | C_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **target_path**: `String` | Reference to an object node using archetype path notation. |
| **Functions** | **Signature** | **Meaning** |
| | **use_target_occurrences**: `Boolean` ***Post***: Result = (occurrences = Void) | True if target occurrences are to be used as the value of occurrences in this object; by the time of runtime use, the target occurrences value has to be set into this object. |

**Validity rules**

The following validity rules applies to internal references.

— **VUNT** use_node reference model type validity: the reference model type mentioned in an `C_COMPLEX_OBJECT_PROXY` node shall be the same as or a super-type (according to the reference model) of the reference model type of the node referred to.

— **VUNP** use_node path validity: the path mentioned in a use_node statement shall refer to an object node defined elsewhere in the same archetype or any of its specialisation parent archetypes, that is not itself an internal reference node, and which carries a node identifier if one is needed at the reference point.

The following validity rule applies to the redefinition of an internal reference in a specialised archetype.

— **VSUNT** use_node specialisation parent validity: a `C_COMPLEX_OBJECT_PROXY` node may be redefined in a specialised archetype by another `C_COMPLEX_OBJECT_PROXY` (e.g. in order to redefine occurrences), or by a `C_COMPLEX_OBJECT` structure that legally redefines the target `C_COMPLEX_OBJECT` node referred to by the reference.

### 7.4.5.11 C_PRIMITIVE_OBJECT class

| **Class** | ***C_PRIMITIVE_OBJECT (abstract)*** | |
|---|---|---|
| **Description** | Parent of types representing constraints on primitive types. | |
| **Inherit** | C_DEFINED_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **assumed_value**: `Any` | Value to be assumed if none sent in data. |
| **0..1** | **is_enumerated_type_constraint**: `Boolean` | True if this object represents a constraint on an enumerated type from the reference model, where the latter is assumed to be based on a primitive type, generally Integer or String. |
| **1..1** | **constraint**: `Any` | Constraint represented by this object; redefine in descendants. |
| **Functions** | **Signature** | **Meaning** |
| | **has_assumed_value**: `Boolean` | True if there is an assumed value. |

| | | |
|---|---|---|
| | **constrained_typename**: `String` | Generate name of native type that is constrained by this C_XXX type. For most types, it is the C_XXX typename without the 'C_', i.e. XXX. E.g. C_INTEGER → Integer. For the date/time types the mapping is different. |

**Validity rules**

The validity rules for `C_PRIMITIVE_OBJECTs` are as follows.

— **VOBAV** object node assumed value validity: the value of an assumed value shall fall within the value space defined by the constraint to which it is attached.

### 7.4.5.12 C_BOOLEAN class

| **Class** | **C_BOOLEAN** | |
|---|---|---|
| **Description** | Constraint on instances of Boolean. Both attributes cannot be set to False, since this would mean that the Boolean value being constrained cannot be True or False. | |
| **Inherit** | C_PRIMITIVE_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1 (redefined)** | **constraint**: `List<Boolean>` | Boolean constraint - a list of Boolean values. |
| **0..1 (redefined)** | **assumed_value**: `Boolean` | |
| **1..1 (redefined)** | **default_value**: `Boolean` | |
| **Functions** | **Signature** | **Meaning** |
| **(effected)** | **prototype_value**: `Boolean` | |

### 7.4.5.13 C_STRING class

| **Class** | **C_STRING** | |
|---|---|---|
| **Description** | Constraint on instances of STRING. | |
| **Inherit** | C_PRIMITIVE_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1 (redefined)** | **constraint**: `List<String>` | String constraint - a list of literal strings and / or regular expression strings delimited by the '/' character. |
| **1..1 (redefined)** | **default_value**: `String` | |
| **0..1 (redefined)** | **assumed_value**: `String` | |
| **Functions** | **Signature** | **Meaning** |
| **(effected)** | **prototype_value**: `String` | |
| **(effected)** | **valid_value** (a_value: `String`): `Boolean` | True if a_value is valid with respect to constraint expressed in concrete instance of this type. |

### 7.4.5.14 C_ORDERED class

| **Class** | ***C_ORDERED (abstract)*** |
|---|---|

| | | |
|---|---|---|
| **Description** | Abstract parent of primitive constrainer classes based on ORDERED base types, i.e. types like Integer, Real, and the Date/Time types. The model constraint is a List of Intervals, which may include point Intervals, and acts as an efficient and formally tractable representation of any number of point values and/or contiguous intervals of an ordered value domain.<br>In its simplest form, the constraint accessor returns just a single point Interval<T> object, representing a single value.<br>The next simplest form is a single proper Interval <T> (i.e. normal two-sided or half-open interval). The most complex form is a list of any combination of point and proper intervals. | |
| **Inherit** | C_PRIMITIVE_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1 (redefined)** | **constraint**: `List<Interval>` | |

### 7.4.5.15 C_INTEGER class

| **Class** | **C_INTEGER** |
|---|---|
| **Description** | Constraint on instances of Integer. |

### 7.4.5.16 C_REAL class

| **Class** | **C_REAL** |
|---|---|
| **Description** | Constraint on instances of Real. |

### 7.4.5.17 C_TEMPORAL class

| **Class** | ***C_TEMPORAL (abstract)*** | |
|---|---|---|
| **Description** | Purpose Abstract parent of C_ORDERED types whose base type is an ISO date/time type. | |
| **Inherit** | C_ORDERED | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1** | **pattern_constraint**: `String` | Optional alternative constraint in the form of a pattern based on ISO8601. See descendants for details. |
| **Functions** | **Signature** | **Meaning** |
| | **valid_pattern_constraint**: `Boolean` | |

### 7.4.5.18 C_DATE class

| **Class** | **C_DATE** | |
|---|---|---|
| **Description** | ISO 8601-compatible constraint on instances of Date in the form either of a set of validity values, or else date ranges based on the C_ORDERED list constraint. There is no validity flag for 'year', since it shall always be by definition mandatory in order to have a sensible date at all. Syntax expressions of instances of this class include "YYYY-??-??" (date with optional month and day). | |
| **Functions** | **Signature** | **Meaning** |
| | **month_validity**: `VALIDITY_KIND` | Validity of month in constrained date. |
| | **day_validity**: `VALIDITY_KIND` | Validity of day in constrained date. |
| | **timezone_validity**: `VALIDITY_KIND` | Validity of timezone in constrained date. |
| **Invariant** | ***Pattern_validity***: pattern /= Void implies valid_iso8601_date_constraint_pattern(pattern) | |

### 7.4.5.19 C_TIME class

| Class | C_TIME | |
|---|---|---|
| Description | ISO 8601-compatible constraint on instances of Time in the form either of a set of validity values, or else date ranges based on the C_ORDERED list constraint. There is no validity flag for 'hour', since it shall always be by definition mandatory in order to have a sensible time at all. Syntax expressions of instances of this class include "HH:??:xx" (time with optional minutes and seconds not allowed). | |
| Functions | Signature | Meaning |
| | **minute_validity**: `VALIDITY_KIND` | Validity of minute in constrained time. |
| | **second_validity**: `VALIDITY_KIND` | Validity of second in constrained time. |
| | **millisecond_validity**: `VALIDITY_KIND` | Validity of millisecond in constrained time. |
| | **timezone_validity**: `VALIDITY_KIND` | Validity of timezone in constrained date. |
| Invariant | ***Pattern_validity***: pattern /= Void implies valid_iso8601_time_constraint_pattern (pattern) | |

### 7.4.5.20 C_DATE_TIME class

| Class | C_DATE_TIME | |
|---|---|---|
| Description | ISO 8601-compatible constraint on instances of Date_Time. There is no validity flag for 'year', since it shall always be by definition mandatory in order to have a sensible date/time at all. Syntax expressions of instances of this class include "YYYY-MM-DDT??:??:??" (date/time with optional time) and "YYYY-MMDDTHH:MM:xx" (date/time, seconds not allowed). | |
| Functions | Signature | Meaning |
| | **month_validity**: `VALIDITY_KIND` | Validity of month in constrained date. |
| | **day_validity**: `VALIDITY_KIND` | Validity of day in constrained date. |
| | **timezone_validity**: `VALIDITY_KIND` | Validity of timezone in constrained date. |
| | **minute_validity**: `VALIDITY_KIND` | Validity of minute in constrained time. |
| | **second_validity**: `VALIDITY_KIND` | Validity of second in constrained time. |
| | **millisecond_validity**: `VALIDITY_KIND` | Validity of millisecond in constrained time. |
| Invariant | ***Pattern_validity***: pattern /= Void implies valid_iso8601_date_time_constraint_pattern(pattern) | |

### 7.4.5.21 C_DURATION class

| Class | C_DURATION | |
|---|---|---|
| Description | | |
| Functions | Signature | Meaning |
| | **years_allowed:**: `Boolean` | True if years are allowed in the constrained Duration. |
| | **months_allowed:**: `Boolean` | True if months are allowed in the constrained Duration. |
| | **weeks_allowed:**: `Boolean` | True if weeks are allowed in the constrained Duration. |
| | **days_allowed**: `Boolean` | True if days are allowed in the constrained Duration. |
| | **hours_allowed**: `Boolean` | True if hours are allowed in the constrained Duration. |
| | **minutes_allowed**: `Boolean` | True if minutes are allowed in the constrained Duration. |

| | | |
|---|---|---|
| | **seconds_allowed**: `Boolean` | True if seconds are allowed in the constrained Duration. |
| | **fractional_seconds_allowed**: `Boolean` | True if fractional seconds are allowed in the constrained Duration. |

### 7.4.5.22 C_TERMINOLOGY_CODE class

| **Class** | **C_TERMINOLOGY_CODE** | |
|---|---|---|
| **Description** | Constrainer type for instances of TERMINOLOGY_CODE. The constraint attribute can contain: * a single at-code * a single ac-code, representing a value-set that is defined in the archetype terminology<br><br>If there is an assumed value for the ac-code case above, the assumed_value attribute contains a single at-code, which shall come from the list of at-codes defined as the internal value set for the ac-code. | |
| **Inherit** | C_PRIMITIVE_OBJECT | |
| **Attributes** | **Signature** | **Meaning** |
| **1..1 (redefined)** | **constraint**: `String` | Type of individual constraint - a single string that can either be a local at-code, or a local ac-code signifying a locally defined value set. If an ac-code, assumed_value may contain an at-code from the value set of the ac-code. |
| **0..1 (redefined)** | **assumed_value**: `TERMINOLOGY_CODE` | |
| **1..1 (redefined)** | **default_value**: `TERMINOLOGY_CODE` | |
| **Functions** | **Signature** | **Meaning** |
| | **value_set_expanded**: `List<String>` | Effective set of at-code values corresponding to an ac-code for a locally defined value set. Not defined for ac-codes that have no local value set. |
| | **value_set_substituted**: `List<Uri>` | For locally defined value sets within individual code bindings: return the term URI(s) substituted from bindings for local at-codes in value_set_expanded. |
| | **value_set_resolved**: `List<TERMINOLOGY_CODE>` | For locally defined value sets within individual code bindings: final set of external codes to which value set is resolved. |
| **(effected)** | **valid_value** (a_value: `TERMINOLOGY_CODE`): `Boolean` | True if a_value is valid with respect to constraint expressed in concrete instance of this type. |
| **(effected)** | **prototype_value**: `TERMINOLOGY_CODE` | A generated prototype value from this constraint object. |

### 7.4.5.23 C_SECOND_ORDER class

| **Class** | ***C_SECOND_ORDER (abstract)*** | |
|---|---|---|
| **Description** | Abstract parent of classes defining second order constraints. | |
| **Attributes** | **Signature** | **Meaning** |
| **0..1** | **members**: `List<ARCHETYPE_CONSTRAINT>` | Members of this second order constrainer. Normally redefined in descendants. |
| **Functions** | **Signature** | **Meaning** |