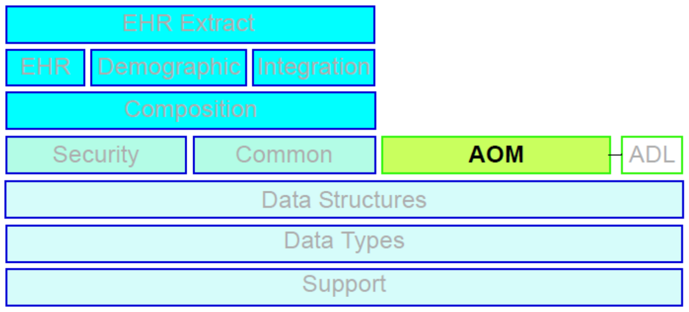
Table of Contents

Archetype Object Model (AOM2)

Issuer: openEHR Specification Program		
Revision: [latest_issue]	**Date**: [latest_issue_date]	**Status**: TRIAL
Keywords: EHR, ADL, health records, archetypes, constraints		



Amendment Record

Issue	Details	Raiser	Completed
2.1.14	Refactor ARCHETYPE and ARCHETYPE_TERMINOLOGY models, in order to simplify: remove differential and flat forms of classes. Split ARCHETYPE into two classes, with AUTHORED_ARCHETYPE as a new class that inherits from AUTHORED_RESOURCE .	T Beale	04 Jan 2015
2.1.13	Remove VDSSR , VSUNC ; add VDSSID , VARXID . Replace '+u' (unstable) version modifier with semver.org standard '-alpha'. Remove overview material to new Archetypes: Technical Overview specification.	T Beale, I McNicoll, S Garde	12 Nov 2014
2.1.12	Remove ARCHETYPE .uid attribute.	H Solbrig	08 Oct 2014
2.1.11	Correct spelling of 'licence' to international English; rename ARCHETYPE.urn to provenance_id.	S Garde, I McNicoll	29 Sep 2014
2.1.10	Modified C_ARCHETYPE_ROOT to have an id-code in all cases. Add error VSONPO , VSONPT : specialised archetype object node prohibited occurrences validity. Added support for constraints on enumerated types.	CIMI, P Langford, T Beale	18 Jul 2014
2.1.9	Convert ARCHETYPE .uid to urn: URN .	I McNicoll, S Garde, T Beale	04 Jun 2014
2.1.8	Rename ARCHETYPE .commit_number to build_count .	I McNicoll, S Garde, T Beale	21 May 2014
2.1.7	Make VACMCL a warning WACMCL .	D Moner	07 Apr 2014
2.1.6	Renamed ARCHETYPE_INTERNAL_REF to C_OBJECT_PROXY .	T Beale	09 Mar 2014
2.1.5	Renamed ontology to terminology and simplified.	T Beale	09 Jan 2014
2.1.4	Remove CONSTRAINT_REF , C_REFERENCE_OBJECT types; introduce new identification system.	T Beale H Solbrig	07 Jan 2014
2.1.3	Detailed Technical Review.	H Solbrig	21 Nov 2013
2.1.2	Remove C_DOMAIN_TYPE ; merge C_PRIMITIVE_OBJECT and C_PRIMITIVE ; Add support for tuple constraints, replacing ADL 1.4 special Ordinal and Quantity constrainer types; Add new primitive type C_TERMINOLOGY_CODE . Added VSONIF , removed VSONCI (dup of VSONI).	H Solbrig T Beale	20 Aug2013

Issue	Details	Raiser	Completed
2.1.1	Remove C_SINGLE_ATTRIBUTE and C_MULTIPLE_ATTRIBUTE classes.	T Beale, S Garde, S Kobayashi, D Moner, T Beale	15 Dec 2011
2.1.0	SPEC-270. Add specialisation semantics to ADL and AOM. Add various attributes and functions to ARCHETYPE_CONSTRAINT descendant classes. * move C_PRIMITIVE.assumed_value to attribute slot in UML * rename C_DEFINED_OBJECT.default_value function to prototype_value * correct assumed_value definition to be like ; remove its entry from all of the C_PRIMITIVE subtypes * convert BOOLEAN flag representation of patterns to functions and add a String data member for the pattern value, thus matching the XSDs and ADL * add ARCHETYPE.is_template attribute. * add ARCHETYPE .is_component attribute. * allow computed as well as stored attributes. * make ONTOLOGY .terminologies_available computed.	T Beale	10 Dec 2009
2.0.9	SPEC-263. Change Date, Time etc classes in AOM to _ISO8601`DATE` , _ISO8601`TIME` etc from Support IM.	T Beale	20 Jul 2009
	SPEC-296. Convert *Interval<Integer>* to MULTIPLICITY_INTERVAL to simplify specification and implementation.	T Beale	
	SPEC-300. Archetype slot regular expressions should cover whole identifier. Added C_STRING.is_pattern .	A Flinton	
	SPEC-303. Make existence, occurrences and cardinality optional in AOM.	S Heard	
	SPEC-308. Add validity rules to ARCHETYPE_TERMINOLOGY . SPEC-309. ARCHETYPE_CONSTRAINT adjustments. SPEC-178. Add template object model to AM. * Add is_exhaustive attribute to ARCHETYPE_SLOT . * Add is_template attribute to ARCHETYPE . * Add terminology_extracts to ARCHETYPE_TERMINOLOGY .	T Beale	
RELEASE 1.0.2			
2.0.2	SPEC-257. Correct minor typos and clarify text. Correct reversed definitions of is_bag and is_set in CARDINALITY class.	C Ma, R Chen, T Cook	20 Nov 2008

Issue	Details	Raiser	Completed
	SPEC-251. Allow both pattern and interval constraint on Duration in Archetypes. Add pattern attribute to `C_DURATION` class.	S Heard	
	R E L E A S E 1.0.1		
2.0.1	CR-000200. Correct Release 1.0 typographical errors. Table for missed class `ASSERTION_VARIABLE` added. Assumed_value assertions corrected; `standard_representation` function corrected. Added missed `adl_version` , `concept` rename from CR-000153.	D Lloyd, P Pazos, R Chen, C Ma	20 Mar 2007
	CR-000216: Allow mixture of W, D etc in ISO8601 Duration (deviation from standard).	S Heard	
	CR-000219: Use constants instead of literals to refer to terminology in RM.	R Chen	
	CR-000232. Relax validity invariant on `CONSTRAINT_REF` .	R Chen	
	CR-000233: Define semantics for `occurrences` on `ARCHETYPE_INTERNAL_REF` .	K Atalag	
	CR-000234: Correct functional semantics of AOM constraint model package.	T Beale	
	CR-000245: Allow term bindings to paths in archetypes.	S Heard	
	R E L E A S E 1.0		
2.0	CR-000153. Synchronise ADL and AOM attribute naming. CR-000178. Add Template Object Model to AM. Text changes only. CR-000167. Add `AUTHORED_RESOURCE` class. Remove `description` package to `resource` package in Common IM.	T Beale	10 Nov 2005
	R E L E A S E 0.96		
0.6	CR-000134. Correct numerous documentation errors in AOM. Including cut and paste error in `TRANSLATION_DETAILS` class in *Archetype* package. Corrected hyperlinks in Section 2.3.	D Lloyd	20 Jun 2005
	CR-000142. Update ADL grammar to support assumed values. Changed `C_PRIMITIVE` and `C_DOMAIN_TYPE` .	S Heard, T Beale	
	CR-000146: Alterations to *am.archetype.description* from CEN MetaKnow	D Kalra	
	CR-000138. Archetype-level assertions.	T Beale	
	CR-000157. Fix names of `OPERATOR_KIND` class attributes	T Beale	

Issue	Details	Raiser	Completed
	RELEASE 0.95		
0.5.1	Corrected documentation error - return type of `ARCHETYPE_CONSTRAINT`.`has_path` add optionality markers to Primitive types UML diagram. Removed erroneous aggregation marker from `ARCHETYPE_ONTOLOGY`.`parent_archetype` and `ARCHETYPE_DESCRIPTION`.`parent_archetype`.	D Lloyd	20 Jan 2005
0.5	CR-000110. Update ADL document and create AOM document. Includes detailed input and review from: * DSTC * CHIME, Uuniversity College London * Ocean Informatics Initial Writing. Taken from ADL document 1.2draft B.	T Beale A Goodchild Z Tun T Austin D Kalra N Lea D Lloyd S Heard T Beale	10 Nov 2004

Trademarks

'openEHR' is a trademark of the openEHR Foundation

'Microsoft' is a trademark of the Microsoft Corporation

Acknowledgements

The work reported in this document was funded by:

- Ocean Informatics (Australia);
- University College London (UCL), Centre for Health Informatics and Multi-professional Education (CHIME).

This specification and its sibling Archetype Definition Language specification have benefited from wide formal and informal input from the openEHR and wider health informatics community. The openEHR Foundation would like to recognise the following people and institutions.

- Koray Atalag, MD, PhD, Sen. Researcher, National Institute for Health Innovation (NIHI), New Zealand
- Linda Bird PhD, IHTSDO
- Diego Boscá, IBIME, Technical University Valencia, VeraTech for Health, Spain
- Rong Chen MD, PhD, Cambio Healthcare Systems, Sweden
- Borut Fabjan, Program Manager, Marand, Slovenia
- Sebastian Garde PhD, Ocean Informatics UK
- Peter Gummer, Ocean Informatics
- Sam Heard MD, Ocean Informatics
- Stan Huff MD, Intermountain Healthcare, UT, USA
- David Ingram PhD, Emeritus Professor of Health Informatics, UCL
- Dipak Kalra MD, PhD, Professor Health Informatics, CHIME, UCL, president Eurorec
- Shinji Kobayashi PhD, Kyoto University EHR research unit
- Bostjan Lah, Architect, Marand, Slovenia
- David Lloyd, CHIME, UCL (ret)
- Ian McNicoll MD, Ocean Informatics UK
- David Moner, IBIME, Technical University Valencia, VeraTech for Health, Spain
- Pablo Pazos Gutierrez, Tarmac IT, CaboLabs, Uruguay
- Harold Solbrig, Mayo Clinic, Rochester, USA
- Erik Sundvall PhD, Linkoping University, Sweden
- Alessandro Torrisi, Code24, The Netherlands.

Chapter 1. Introduction

1.1. Purpose

This document contains the normative description of openEHR Archetype and Template semantics (originally described in [Beale_2000] and [Beale_2002]), 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, such as the openEHR Archetype Definition Language (ADL), XML and so on.

It is recommended in any case that the ADL specification be read in conjunction with this document, since it contains a detailed explanation of the semantics of archetypes, and many of the examples are more obvious in ADL, regardless of whether ADL is actually used with the object model presented here or not.

The release of AOM described in this specification corresponds to the 2.x version of the archetype formalism.

1.2. Related Documents

Prerequisite documents for reading this document include:

- The openEHR Architecture Overview
- The openEHR Archetypes: Technical Overview specification

Related documents include:

- The openEHR Archetype Definition Language (ADL)
- The openEHR Operational Template Specification

1.3. 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’.

We also use the word ‘archetype’ in a broad sense to designate what are commonly understood to be ‘archetypes’ (specifications of clinical data groups / data constraints) and ‘templates’ (data sets based on archetypes, since at a technical level, an ADL/AOM 2 template is in fact just an archetype. Accordingly, statements about ‘archetypes’ in this specification can be always understood to also apply to templates, unless otherwise indicated.

1.4. Status

This specification is in the ‘development’ state, and is published for review purposes.

The development version of this document can be found at *http://www.openehr.org/releases/trunk/architecture/am/aom2.pdf* .

Blue text indicates sections under active development.

1.5. Tools

Various tools exist for creating and processing archetypes. The openEHR tools are available in source and binary form from the website (*http://www.openEHR.org*).

1.6. Changes from Previous Versions

1.6.1. Release 1.5 to 2.0 (Document version 2.1.2 -)

The changes in release 2 of the ADL/AOM formalism are designed to make the formalism more computable with respect to terminology, and enable more rigorous validation and flattening operations.

The changes 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, **namespaced identifier**;
- Addition of **explicit value-sets** in terminology section, replacing in-line value sets in the `definition` section;
- Renaming archetype `ontology` section to `terminology`;
- Expression of all external **term bindings as URIs** following IHTSDO format;
- Introduction of **‘tuple’ constraints** to replace openEHR custom constrainer types for covarying attributes within Quantity, Ordinal structures;
- Re-engineering of all primitive constrainer types, i.e. `C_STRING` , `C_DATE` etc;
- Removal of the openEHR Archetype Profile specification.

1.6.2. Release 1.4 to 1.5 (Document version 2.0 to 2.1.1)

The changes in release 1.5 are made to better facilitate the representation of specialised archetypes. The key semantic capability for specialised archetypes is to be able to support a differential representation, i.e. to express a specialised archetype only in terms of the changed or new elements in its defnition, rather than including a copy of unchanged elements. Doing the latter is clearly unsustainable in terms of change management.

The changes are as follows.

- 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.

1.6.3. Release 0.6 to 1.4

Changes made from Release 1.3 to 1.4:

- added a new attribute `adl_version` : `String` to the `ARCHETYPE` class;
- changed name of `ARCHETYPE` . `concept_code` attribute to `concept` .

Chapter 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. It is dependent on the openEHR Base package classes.

2.1. Package Structure

The openEHR Archetype Object Model is defined as the package `am.archetype` , as illustrated in Figure . It is shown in the context of the openEHR `am.archetype` packages.

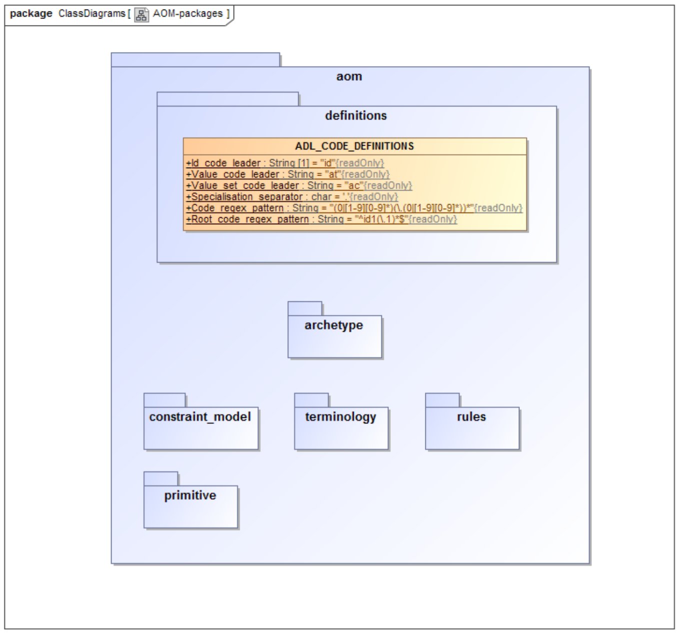
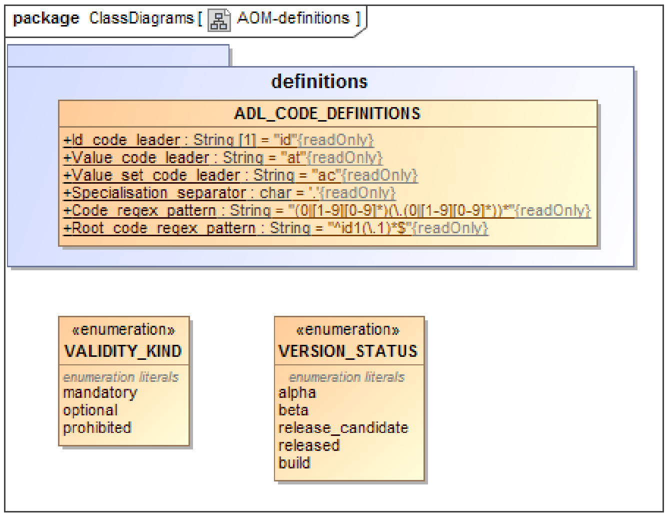

package ClassDiagrams [AOM-packages]
aom
definitions
ADL_CODE_DEFINITIONS
+Id_code_leader : String [1] = "id"{readOnly}
+Value_code_leader : String = "at"{readOnly}
+Value_set_code_leader : String = "ac"{readOnly}
+Specialisation_separator : char = '.'{readOnly}
+Code_regex_pattern : String = "(0|[1-9][0-9]*)(\.(0|[1-9][0-9]*))*"{readOnly}
+Root_code_regex_pattern : String = "^id1(\.1)*$"{readOnly}
archetype
constraint_model
terminology
rules
primitive

Figure 1. Package Overview

Chapter 3. Definition and Utility Classes

3.1. Overview

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



package ClassDiagrams [AOM-definitions]
definitions
ADL_CODE_DEFINITIONS
+Id_code_leader : String [1] = "id"{readOnly}
+Value_code_leader : String = "at"{readOnly}
+Value_set_code_leader : String = "ac"{readOnly}
+Specialisation_separator : char = '.'{readOnly}
+Code_regex_pattern : String = "(0|[1-9][0-9]*)(\.(0|[1-9][0-9]*))*"{readOnly}
+Root_code_regex_pattern : String = "^id1(\.1)*$"{readOnly}
«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` (openEHR `resource` package) and its subordinate classes. These are shown in full within the packages that use them.

3.2. Class Definitions

3.2.1. 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”.

3.2.2. VALIDITY_KIND Enumeration

Enumeration	**VALIDITY_KIND**	
Description	An enumeration of three values that may commonly occur in constraint models. 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.

3.2.3. 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 archteype 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..

Chapter 4. The Archetype Package

4.1. Overview

The top-level model of archetypes and templates (all variant forms) is illustrated in Figure 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, 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 (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.

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.

4.2. Archteype Identification

4.2.1. 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 the following diagram, which also shows the computed `semantic_id` and `physical_id` forms.

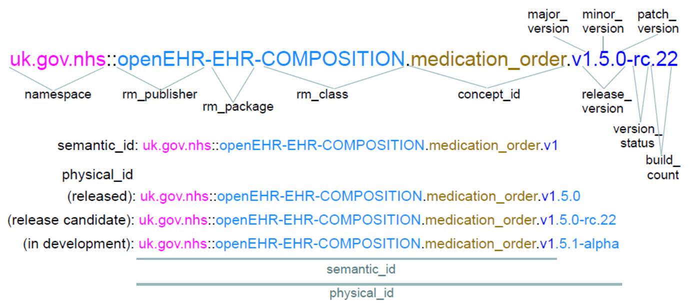

major_ version
minor_ version
patch_ version
uk.gov.nhs::openEHR-EHR-COMPOSITION.medication_order.v1.5.0-rc.22
namespace
rm_publisher
rm_package
rm_class
concept_id
release_ version
version_ status
build_ count
semantic_id: uk.gov.nhs::openEHR-EHR-COMPOSITION.medication_order.v1
physical_id
(released): uk.gov.nhs::openEHR-EHR-COMPOSITION.medication_order.v1.5.0
(release candidate): uk.gov.nhs::openEHR-EHR-COMPOSITION.medication_order.v1.5.0-rc.22
(in development): uk.gov.nhs::openEHR-EHR-COMPOSITION.medication_order.v1.5.1-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 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). A full explanation of the identification system and rules is given in the openEHR Knowledge Artefact Identification specification.

4.2.2. 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.

4.3. Top-level Meta-data

The following items correspond to syntax elements that may appear in parentheses in the first line of an ADL archetype.

4.3.1. ADL Version

The `ARCHETYPE.adl_version` attribute in ADL 1.4 was used to indicate the ADL release used in the archetype source file from which the AOM structure was created (the version number comes from the revision history of the openEHR ADL specification). In the current and future AOM and ADL specifications, the meaning of this attribute is generalised to mean ‘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.

4.3.2. 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 archteypes.

4.3.3. Generated Flag

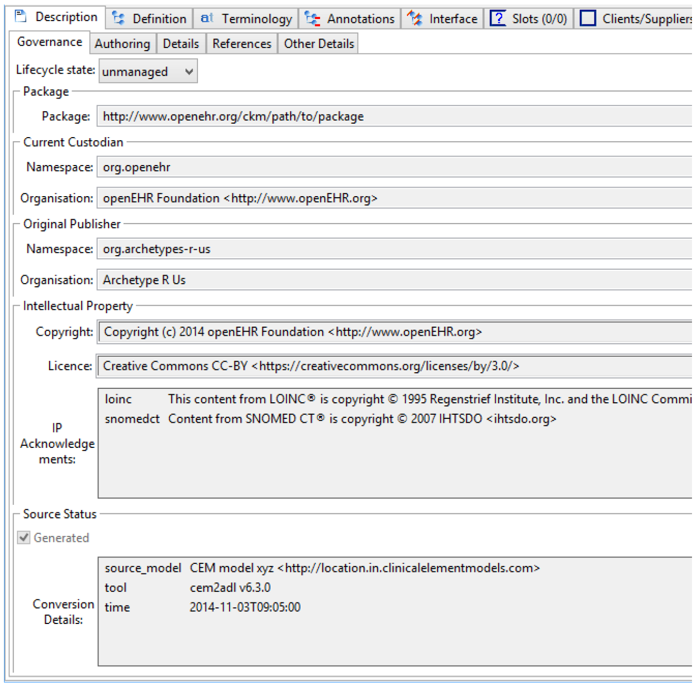
The `ARCHETYPE.is_generated` flag is used to indicate that an archetype has been machine-generated from another artefact, e.g. an older ADL version (say 1.4), or a non-archetype artefact. If true, it indicates to tools that the current archetype can potentially be overwritten, and that some other artefact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

4.4. 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.

4.4.1. 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. A typical form of these is shown in the screenshot in Figure Governance Meta-data.



Lifecycle state: unmanaged

Package

Package: http://www.openehr.org/ckm/path/to/package

Current Custodian

Namespace: org.openehr

Organisation: openEHR Foundation <http://www.openEHR.org>

Original Publisher

Namespace: org.archetypes-r-us

Organisation: Archetype R Us

Intellectual Property

Copyright: Copyright (c) 2014 openEHR Foundation <http://www.openEHR.org>

Licence: Creative Commons CC-BY <https://creativecommons.org/licenses/by/3.0/>

IP Acknowledgements:

loinc	This content from LOINC® is copyright © 1995 Regenstrief Institute, Inc. and the LOINC Commi
snomedct	Content from SNOMED CT® is copyright © 2007 IHTSDO <ihtsdo.org>

Source Status

☑ Generated

Conversion Details:

source_model	CEM model xyz <http://location.in.clinicalelementmodels.com>
tool	cem2adl v6.3.0
time	2014-11-03T09:05:00

Figure 5. Governance Meta-data

Package

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. The lifecycle state machine and versioning rules are explained fully in the openEHR Knowledge Artefact Identification specification. Here we simply note that the value of the property is a coded term corresponding to one of the macro-state names on the diagram, i.e. ‘unmanaged’, ‘in_development’, and so on.

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 publisherof the artefact, if there is one.

Intellectual Property Items

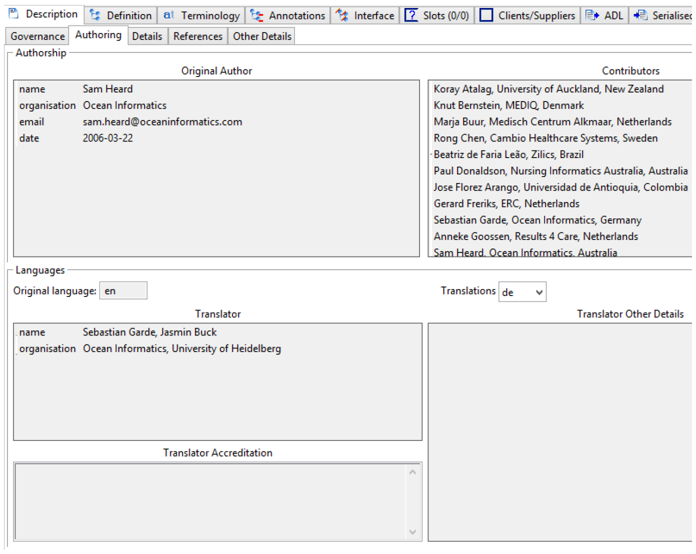
There are three properties in the class that `RESOURCE_DESCRIPTION` relate to intellectual property (IP). Licence is a String field for recording of the licence (US: ‘license’) under which the artefact can be used. The recommended format is

```
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).

4.4.2. Authorship Meta-data

Authorship meta-data consists of items such as author name, contributors, and translator information, and is visualised in Figure Authoring Meta-data.



Original Author

name	Sam Heard
organisation	Ocean Informatics
email	sam.heard@oceaninformatics.com
date	2006-03-22

Contributors

Koray Atalag, University of Auckland, New Zealand
Knut Bernstein, MEDIQ, Denmark
Marja Buur, Medisch Centrum Alkmaar, Netherlands
Rong Chen, Cambio Healthcare Systems, Sweden
Beatriz de Faria Leão, Zilics, Brazil
Paul Donaldson, Nursing Informatics Australia, Australia
Jose Florez Arango, Universidad de Antioquia, Colombia
Gerard Freriks, ERC, Netherlands
Sebastian Garde, Ocean Informatics, Germany
Anneke Goossen, Results 4 Care, Netherlands
Sam Heard, Ocean Informatics, Australia

Original language: en

Translations: de

Translator

name	Sebastian Garde, Jasmin Buck
organisation	Ocean Informatics, University of Heidelberg

Figure 6. Authoring Meta-data

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

Languages and Translation

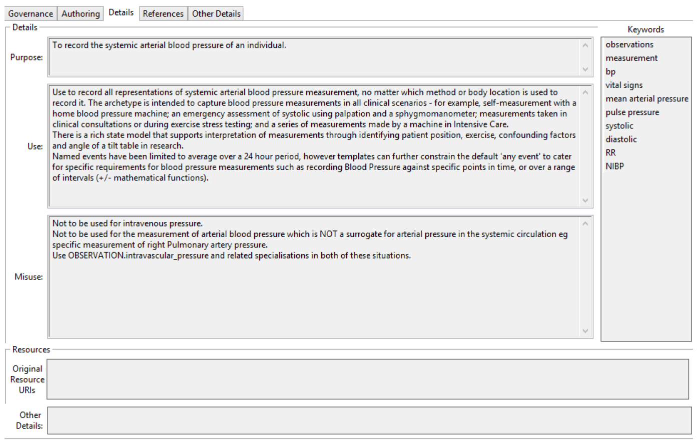
The `AUTHORED_RESOURCE`.`original_language` and `TRANSLATION_DETAILS` class enable the original language of authoring and information relating to subsequent translations to be expressed. `TRANSLATION_DETAILS` .`author` allows each translator to be represented in the same way as the `original_author` , 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 may be out of date.

4.4.3. 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, as shown in Figure Descriptive Meta-data.



Purpose: To record the systemic arterial blood pressure of an individual.

Use: Use to record all representations of systemic arterial blood pressure measurement, no matter which method or body location is used to record it. The archetype is intended to capture blood pressure measurements in all clinical scenarios - for example, self-measurement with a home blood pressure machine; an emergency assessment of systolic using palpation and a sphygmomanometer; measurements taken in clinical consultations or during exercise stress testing; and a series of measurements made by a machine in Intensive Care.
There is a rich state model that supports interpretation of measurements through identifying patient position, exercise, confounding factors and angle of a tilt table in research.
Named events have been limited to average over a 24 hour period, however templates can further constrain the default 'any event' to cater for specific requirements for blood pressure measurements such as recording Blood Pressure against specific points in time, or over a range of intervals (+/- mathematical functions).

Misuse: Not to be used for intravenous pressure.
Not to be used for the measurement of arterial blood pressure which is NOT a surrogate for arterial pressure in the systemic circulation eg specific measurement of right Pulmonary artery pressure.
Use OBSERVATION.intravascular_pressure and related specialisations in both of these situations.

Keywords: observations, measurement, bp, vital signs, mean arterial pressure, pulse pressure, systolic, diastolic, RR, NIBP

Original Resource URIs:

Other Details:

Figure 7. Descriptive Meta-data

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 may not be useful, since 'resources' are not typically available for each language.

4.5. Structural Definition

4.5.1. 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 appopriate `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 may 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.

4.5.2. Structural Variants

The model in Figure 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 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.

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 chapter of this specification, (also in the ADL 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 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.

Another archetype variant, also shown in 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 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.

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

4.6. Class Descriptions

4.6.1. 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 par-ent 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	

4.6.2. 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	
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 may 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)	

4.6.3. 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 sytem.
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)	

4.6.4. 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 from the owning template.
Invariant	***Inv_is_specialised***: is_specialised	

4.6.5. 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 docu-mented by their owning template.
Inherit	ARCHETYPE
Invariant	***Inv_is_specialised***: is_specialised

4.6.6. 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. An operational template is used for generating and validating canonical openEHR 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	

4.7. 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 must exist and consist of a valid 3-part version identifier.

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

VARCN: archetype concept validity. The node_id of the root object of the archetype must be of the form id1\{.1}*, where the number of ‘.1’ components equals the specalisation depth, and must be defined in the terminology.

VATDF: value code validity. Each value code (at-code) used in a term constraint in the archetype definition must 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 must 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 must 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 must exist in the relevant terminology (subject to tool accesibility; codes for inaccessible terminologies should be flagged with a warning indicating that no verification was possible).

VOTM: terminology translations validity. Translations must 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 desription, terminology, and annotations sections, each item must 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 must 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 must 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 must 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 must have an identifier that conforms to the openEHR specification for archetype identifiers.

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

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

The following rules apply to specialised archetypes.

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

VALC: archetype language conformance. The languages defined in a specialised archetype must 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 must 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 must have a specialisation level no greater than the specialisation level of the archetype.

Chapter 5. Constraint Model Package

5.1. Overview

Figure and Figure 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.

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.

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.

5.2. Semantics

The effect of the model is to create archetype description structures that are a hierarchical alternation of object and attribute constraints. This structure can be seen by inspecting an ADL archetype, or by viewing an archetype in the openEHR ADL workbench See openEHR. EHR Reference Model. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/top.html., and is a direct consequence of the object-oriented principle that classes consist of properties, which in turn have types that are classes. (To be completely correct, types do not always correspond to classes in an object model, but it does not make any difference here). The repeated object/attribute hierarchical structure of an archetype provides the basis for using paths to reference any node in an archetype. Archetype paths follow a syntax that is a directly convertible in and out of the W3C Xpath syntax.

5.2.1. All Node Types

5.3. 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.

5.4. 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 must 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 may 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. The formal semantics of both functions are given in the section See Class Definitions.

5.4.1. Attribute Node Types

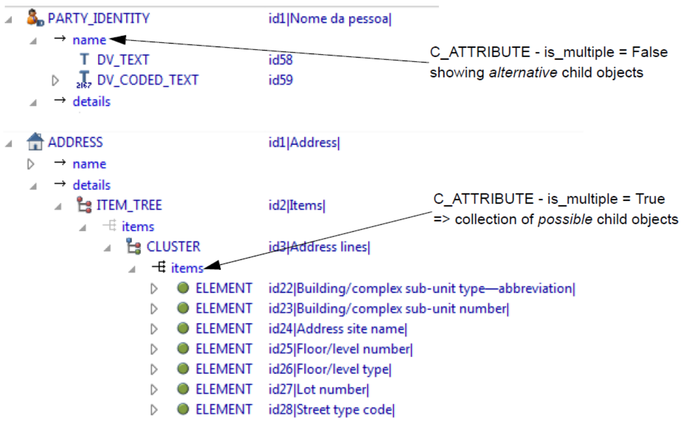
Constraints on reference model attributes, including computed attributes (represented by functions with no aguments in most programming languages), are represented by instances of `C_ATTRIBUTE` . The expressible constraints include:

- `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) must 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 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 must 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.



```
PARTY_IDENTITY                id1|Nome da pessoa|
  → name
      T DV_TEXT               id58
      DV_CODED_TEXT           id59
  → details

ADDRESS                       id1|Address|
  → name
  → details
      ITEM_TREE               id2|Items|
        items
          CLUSTER             id3|Address lines|
            items
              ELEMENT   id22|Building/complex sub-unit type—abbreviation|
              ELEMENT   id23|Building/complex sub-unit number|
              ELEMENT   id24|Address site name|
              ELEMENT   id25|Floor/level number|
              ELEMENT   id26|Floor/level type|
              ELEMENT   id27|Lot number|
              ELEMENT   id28|Street type code|
```

C_ATTRIBUTE - is_multiple = False
showing *alternative* child objects

C_ATTRIBUTE - is_multiple = True
=> collection of *possible* child objects

Figure 13. C_ATTRIBUTE variants

5.4.2. 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 must 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 within 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.

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 characterise `C_DEFINED_OBJECT` s 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 archteype-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 must 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.

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.

'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.

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` .

Any_allowed

The any_allowed function on a node 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.

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 covarying 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`	A string containing either a single at-code or a single ac-code. In the latter case, the constraint represents either a locally defined value set or (via 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	`T → Integer`	As for Ordered type, with T = `Integer`
Real	`T → Real`	As for Ordered type, with T = `Real`
Temporal types	`List <Interval <T→ISO8601_TYPE>>` OR `String (pattern)`	As for ordered types, with T being an ISO8601-based type, with the addition of a second type constraint - a pattern based on ISO8601 syntax.,
Date	`T → ISO8601_DATE`	As for Temporal types with T = `ISO8601_DATE`
Time	`T → ISO8601_TIME`	As for Temporal types with T = `ISO8601_TIME`
Date_time	`T → ISO8601_DATE_TIME`	As for Temporal types with T = `ISO8601_DATE_TIME`
Duration	`T → ISO8601_DURATION`	As for Temporal types with T = `ISO8601_DURATION`

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 the first case, an additional at-code may be included as an assumed value; the at-code must come from the locally defined value set;

- a single at-code, repesenting a single possible value. In theory this could 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.

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 correctly within software). 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.

Assumed_value

The 'assumed_value' concept 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.

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.

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.

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 stricly speaking UML doesn't 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 of a type enumeration.

enumerated type constraint | diagrams/enumerated_type_constraint.svg

Figure 14. Enumerated Constraint

A parser that deserialises 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.

5.5. 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 ('invariants' in ADL/AOM 1.4) 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 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.

5.5.1. 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 below.

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 may 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 a 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.

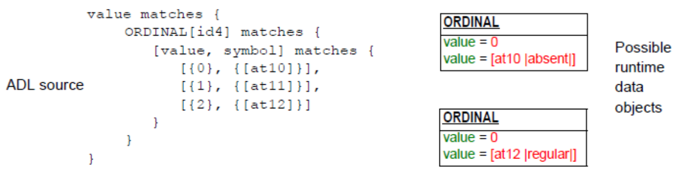

```
value matches {
    ORDINAL[id4] matches {
        [value, symbol] matches {
            [{0}, {[at10]}],
            [{1}, {[at11]}],
            [{2}, {[at12]}]
        }
    }
}
```

Figure 17. Tuple Constraint Example Data

5.5.2. 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. Examples of archetype slots in ADL syntax are given in the openEHR ADL document.

5.6. AOM Type Substitutions

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

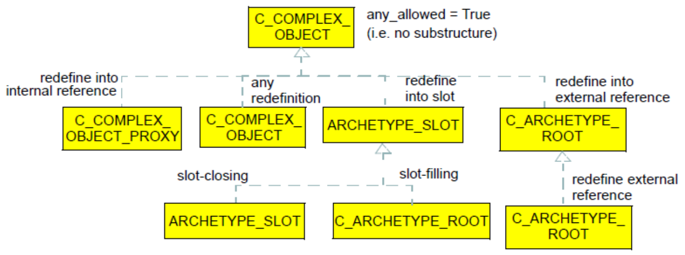


Figure 18. C_Object Substitutions

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 may 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.

5.7. Class Definitions

5.7.1. ARCHETYPE_CONSTRAINT Class

Class	***ARCHETYPE_CONSTRAINT (abstract)***	
Description	Archetype equivalent to LOCATABLE class in openEHR Common reference model. Defines common constraints for any inheritor of LOCATABLE in any reference model.	
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 special-ised 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 redefnition 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	

5.7.2. 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` ***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` ***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 special-ised archetype nodes.

5.7.3. 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
```

5.8. Validity Rules

The validity rules are as follows:

VCARM: attribute name reference model validity: an attribute name introducing an attribute constraint block must 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, must 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 must 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 must 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, must 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 must 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 must 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 must 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 must 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, must conform to the cardinality of the corresponding node in the flat parent archetype by either being identical, or being wholly contained by the latter.

5.8.1. 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.

5.8.2. 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 dis-tinguish sibling nodes. All nodes must have a node_id; for nodes under a container C_ATTRIBUTE, the id must be an id-code must be defined in the archetype terminolo-gy. 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.

5.9. Occurrences inferencing rules

The notion of 'occurrences' does not exist in an object model that might be used as the reference model on which archteypes 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.

5.10. 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
```

5.11. 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 must be defined in the underlying information model.

VCORMT object constraint type validity: a type name introducing an object constraint block must 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 must have a node identifier.

VCOSU object node identifier validity: every object node must 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 must 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 must 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.

Deprecated: **VSONIR** specialised archetype redefined object node identifier condition: the node identifier of an object node in a specialised archetype that is a redefinition of a node in the flat parent must be redefined if any of reference model type, node identifier definition in the terminology, or occurrences of the immediate object constraint is redefined, with the exception of occurrences being redefined to {0}, i.e. exclusion.

Deprecated: **VSONI** specialised archetype redefined object node identifier validity: if an object node in a specialised archetype is a redefinition of a node in the flat parent according to VSONIR, and the parent node carries a node identifier, it must carry a node identifier specalised at the level of the child archetype. Otherwise it must carry the same node identifier (or none) as the corresponding parent node.

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 must 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 must 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, must 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} must 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 must refer to a node found within the same container in the flat parent archetype.

5.11.1. 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. 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.

5.11.2. 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.

5.11.3. 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.

5.12. Validity Rules

The validity rules for `C_COMPLEX_OBJECTs` are as follows:

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

5.12.1. 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. 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. In all uses within source archetypes and templates, the children attribute is Void. 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. Typi-cally an ‘interface’ archetype id, i.e. identifi-er 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 must 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 must 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 must 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 must 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 must 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 must have a node id that is a specialisation of that of the slot.

5.12.2. 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 must conform to the published openEHR specification for archetype identifiers.

VDSIV archetype slot 'include' constraint validity. The 'include' constraint in an archetype slot must conform to the slot constraint validity rules.

VDSEV archetype slot 'exclude' constraint validity. The 'exclude' constraint in an archetype slot must 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 must 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 must 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 must 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.

5.12.3. 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 seri- alised 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 must 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 must 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.

5.12.4. 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 must fall within the value space defined by the constraint to which it is attached.

5.12.5. 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`	

5.12.6. 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.

5.12.7. 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 a efficient and formally tractable representation of any number of point values and/or contiguous intervals of an ordered value domain. In its simplest form, the constraint accessor returns just a single point Interval<T> object, representing a single value. 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<T>>`	

5.12.8. C_INTEGER Class

Class	C_INTEGER
Description	Constraint on instances of Integer.

5.12.9. C_REAL Class

Class	C_REAL
Description	Constraint on instances of Real.

5.12.10. 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`	

5.12.11. 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 must 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)

5.12.12. 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 must 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)	

5.12.13. 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 must 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)

5.12.14. 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.

5.12.15. C_TERMINOLOGY_CODE Class

Class	**C_TERMINOLOGY_CODE**	
Description	Constrainer type for instances of TERMINOLOGY_CODE. The primary expression of the constraint is in the property `tuple_constraint', and comes in 3 variations: * a single at-code * a single ac-code, representing a value-set that is defined in the archetype terminology * a list of at- and/or ac-codes, representing the possibilities of a tuple constraint The last possibility above is enabled by the merge_tuple routine, which enables the constraint of another single-valued C_TERMINOLOGY_CODE to be merged with the current one.	
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 value or value set of single constraint in tuple_constraint, mediated by terminology to expand an ac-code.
	value_set_substituted: List<Uri>	List of external URI(s) either substituted for local at-codes in value_set_expanded, or else an external ref-set URI substituted for the accode in value_set_id, via bindings, if they exist.
	value_set_resolved: List<TERMINOLOGY_CODE>	Final set of codes, which may be internal or external, to which value set is resolved. For internally defined value sets, the list is 1:1 with value_set_substituted. For external value-sets, the list is determined by a terminology service.
(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.

5.12.16. TERMINOLOGY_CODE Class

Class	**TERMINOLOGY_CODE**	
Description	Logically primitive type representing a reference to a terminology concept, in the form of a terminology identifier, optional version, and a code or code string from the terminology.	
Attributes	**Signature**	**Meaning**
1..1	**terminology_id**: String	
0..1	**terminology_version**: String	
1..1	**code_string**: String	
0..1	**uri**: Uri	

5.12.17. 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.

5.12.18. C_ATTRIBUTE_TUPLE Class

Class	**C_ATTRIBUTE_TUPLE**	
Description	Object representing a constraint on an atttribute tuple, i.e. a group of attributes that are constrained together. Typically used for representing co-varying constraints like {units, range} constraints.	
Inherit	C_SECOND_ORDER	
Attributes	**Signature**	**Meaning**
0..1	**tuples**: `List<C_PRIMITIVE_TUPLE>`	Tuple definitions.
0..1 (redefined)	**members**: `List<C_ATTRIBUTE>`	List of C_ATTRIBUTEs forming the definition of the tuple.

Chapter 6. The Assertion Package

6.1. Overview

Assertions are expressed in archetypes in typed first-order predicate logic (FOL). They are used in two places: to express archetype slot constraints, and to express rules in complex object constraints. In both of these places, their role is to constrain something inside the archetype. Constraints on external resources such as terminologies are expressed in the constraint binding part of the archetype terminology, described in Figure Terminology Package. The assertion package is illustrated below in Rules Package.

6.2. Semantics

Archetype assertions are statements which contain the following elements:

- variables, which are inbuilt, archetype path-based, or external query results;
- manifest constants of any primitive type, including the date/time types
- arithmetic operators: +, *, -, /, ^ (exponent), % (modulo division)
- relational operators: >, <, >=, ⇐, =, !=, matches
- boolean operators: not, and, or, xor
- quantifiers applied to container variables: for_all, exists

A syntax of assertions is defined in the openEHR ADL specification. The package described here is designed to allow the representation of a general-purpose expression tree, as generated by a parser. This relatively simple model of expressions is sufficiently powerful for representing the subset of FOL expressions required in archetypes and templates.

6.3. Class Descriptions

6.3.1. RULE_STATEMENT Class

Class	***RULE_STATEMENT (abstract)***
Description	Abstract concept of any statement in a block of rule statements.
Inherit	RULE_ELEMENT

6.3.2. ASSERTION Class

Class	**ASSERTION**

Description	Structural model of a typed first order predicate logic assertion, in the form of an expression tree, including optional variable definitions.	
Inherit	RULE_STATEMENT	
Attributes	**Signature**	**Meaning**
0..1	**tag**:	Expression tag, used for differentiating multiple assertions.
0..1	**string_expression**:	String form of expression, in case an expression evaluator taking String expressions is used for evaluation.
0..1	**variables**: `List<ASSERTION_VARIABLE>`	Definitions of variables used in the assertion expression.
1..1	**expression**: `EXPR_ITEM`	Root of expression tree.

6.3.3. VARIABLE_DECLARATION Class

Class	***VARIABLE_DECLARATION (abstract)***	
Description	Definition of a named variable used in an assertion expression.	
Inherit	RULE_STATEMENT	
Attributes	**Signature**	**Meaning**
1..1	**name**: `String`	
Functions	**Signature**	**Meaning**
	definition: `String`	Formal definition of the variable.
	value: `Any`	Formal definition of the variable. Value of the variable once evaluated.

6.3.4. EXPR_VARIABLE Class

Class	**EXPR_VARIABLE**	
Description	A variable whose definition is an expression, including atomic expressions such as constants and model references (i.e. path references).	
Inherit	VARIABLE_DECLARATION	
Attributes	**Signature**	**Meaning**
1..1	**expression**: `EXPR_ITEM`	Expression tree of expression.

6.3.5. BUILTIN_VARIABLE Class

Class	BUILTIN_VARIABLE
Description	A variable with a name and definition from a small set of assumed environmental variables. It is assumed that the implementation will correctly generate the appropriate values and types for these variables. The current set of built-in varia-bles is as follows: • current_date: ISO8601_DATE • current_time: ISO8601_TIME • current_date_time: ISO8601_DATE_TIME
Inherit	VARIABLE_DECLARATION

6.3.6. QUERY_VARIABLE Class

Class	QUERY_VARIABLE	
Description	Definition of a variable whose value is derived from a query run on a data con-text in the operational environment. Typical uses of this kind of variable are to obtain values like the patient date of birth, sex, weight, and so on. It could also be used to obtain items from a knowledge context, such as a drug database.	
Inherit	VARIABLE_DECLARATION	
Attributes	**Signature**	**Meaning**
1..1	**context**: `String`	Optional name of context. This allows a basic separation of query types to be done in more sophisticated environments. Possible values might be “patient”, “medications” and so on. Not yet standardised.
1..1	**query_id**: `String`	Identifier of query in the external context, e.g. “date_of_birth”. Not yet standardised.
0..1	**query_args**: `List<String>`	Optional arguments to query. Not yet standardised.

6.3.7. EXPR_ITEM Class

Class	*EXPR_ITEM (abstract)*	
Description	Abstract parent of all expression tree items.	
Inherit	RULE_ELEMENT	
Functions	**Signature**	**Meaning**
	value: `Any`	

6.3.8. EXPR_LEAF Class

Class	EXPR_LEAF	
Description	Expression tree leaf item representing one of: • a manifest constant of any primitive type; • a path referring to a value in the archetype; • a constraint; • a variable reference.	
Inherit	EXPR_ITEM	
Functions	**Signature**	**Meaning**
	reference_type: `String`	Type of reference: “constant”, “attribute”, “function”, “constraint”. The first three are used to indicate the referencing mechanism for an operand. The last is used to indicate a constraint operand, as happens in the case of the right-hand operand of the ‘matches’ operator.

6.3.9. EXPR_CONSTANT Class

Class	EXPR_CONSTANT	
Description	Constant expression tree leaf item. This can represent a manifest constant of any primitive type, i.e.: • Integer, • Real, • Boolean, • String, • Character, • Date, • Time, • Date_time, • Duration • an Interval of any of the above types that are Ordered (see Support IM) • a list of any of the above types.	
Inherit	EXPR_LEAF	
Attributes	**Signature**	**Meaning**
1..1	**value**: `Any`	

6.3.10. EXPR_CONSTRAINT Class

Class	EXPR_CONSTRAINT

Description	Expression tree leaf item representing a constraint on a primitive type, expressed in the form of concrete subtype of C_PRIMITIVE_OBJECT.	
Inherit	EXPR_LEAF	
Attributes	**Signature**	**Meaning**
1..1	**item**: `C_PRIMITIVE_OBJECT`	The constraint.

6.3.11. EXPR_ARCHETYPE_ID_CONSTRAINT Class

Class	**EXPR_ARCHETYPE_ID_CONSTRAINT**	
Description	Expression tree leaf item representing a constraint on an archetype identifier.	
Inherit	EXPR_CONSTRAINT	
Attributes	**Signature**	**Meaning**
1..1 (redefined)	**item**: `C_STRING`	

6.3.12. EXPR_MODEL_REF Class

Class	**EXPR_MODEL_REF**	
Description	Expression tree leaf item representing a reference to a value found in data at a location specified by a path in the archetype definition. • A path referring to a value in the archetype (paths with a leading '/' are in the definition section. • Paths with no leading '/' are in the outer part of the archetype, e.g. "arche-type_id/value" refers to the String value of the archetype_id attribute of the enclosing archetype.	
Inherit	EXPR_LEAF	
Attributes	**Signature**	**Meaning**
1..1	**path**: `String`	The path to the archetype node.

6.3.13. EXPR_VARIABLE_REF Class

Class	**EXPR_VARIABLE_REF**	
Description	Expression tree leaf item representing a reference to a defined variable.	
Inherit	EXPR_LEAF	
Attributes	**Signature**	**Meaning**
1..1	**declaration**: `VARIABLE_DECLARATION`	The variable referred to.

6.3.14. EXPR_OPERATOR Class

Class	*EXPR_OPERATOR (abstract)*	
Description	Abstract parent of operator types.	
Inherit	EXPR_ITEM	
Attributes	**Signature**	**Meaning**
0..1	**precedence_overridden**: `Boolean`	True if the natural precedence of operators is overridden in the expression represented by this node of the expression tree. If True, parentheses should be introduced around the totality of the syntax expression corresponding to this operator node and its operands.
1..1	**operator**: `OPERATOR_KIND`	Type of operator as an Integer value from the OPERATOR_KIND enumeration.

6.3.15. EXPR_UNARY_OPERATOR Class

Class	**EXPR_UNARY_OPERATOR**	
Description	Unary operator expression node.	
Inherit	EXPR_OPERATOR	
Attributes	**Signature**	**Meaning**
1..1	**operand**: `EXPR_ITEM`	Operand node.

6.3.16. EXPR_BINARY_OPERATOR Class

Class	**EXPR_BINARY_OPERATOR**	
Description	Binary operator expression node.	
Inherit	EXPR_OPERATOR	
Attributes	**Signature**	**Meaning**
1..1	**left_operand**: `EXPR_ITEM`	Left operand node.
1..1	**right_operand**: `EXPR_ITEM`	Right operand node.

6.3.17. OPERATOR_KIND Enumeration

Enumeration	**OPERATOR_KIND**
Description	Enumeration type for operator types in assertion expressions.

Attributes	Signature	Meaning
	op_eq	Equals operator (= or ==)
	op_ne	Not equals operator (!= or /=)
	op_le	Less-than or equals operator (⇐)
	op_lt	Less-than operator (⇐)
	op_ge	Greater-than or equals operator (>=)
	op_gt	Greater-than operator (>)
	op_matches	Matches operator (matches or is_in)
	op_not	Not logical operator
	op_and	And logical operator
	op_or	Or logical operator.
	op_xor	Xor logical operator
	op_implies	Implies logical operator
	op_for_all	For-all (universal) quantifier
	op_exists	Exists quantifier
	op_plus	Arithmetic plus operator (+)
	op_minus	Arithmetic minus operator (-)
	op_multiply	Arithmetic multiplication operator (*)
	op_divide	Arithmetic division operator (/)
	op_exponent	Arithmetic exponentiation operator (^)

Chapter 7. Terminology Package

7.1. Overview

All local terminology as well as terminological and terminology binding elements of an archetype are represented in the terminology section of an archetype, whose semantics are defined by the *archetype.terminology* package, shown below.

An archetype terminology consists of the following elements.

- term_definitions: a mandatory structure consisting of lists of term definitions defined local to the archetype, one list for each language of translation, as well as the original language of definition. The entries in this table include:
- Some or all id-codes. One of these is a code of the form 'id1', 'id1.1', 'id1.1.1' etc, denoting the concept of the archetype as a whole. This particular code is recorded in the `concept_code` attribute and is used as the id-code on the root node in the archetype definition. Not all id-codes are required to be in the term_definitions structure - for nodes that are children of single-valued attribute, a term definition is optional (and not typically defined).
- at-codes used to define value terms and inline value sets/ All at-codes will appear within a `C_TERMINOLOGY_CODE` constraint object within the archetype. All at-codes must have a definition in the term_definitions.
- ac-codes used to define external value set references. All ac-codes must have a definition in the term_definitions.
- term_bindings: an optional structure consisting of list of terms and bindings, one list for each external terminology (i.e. the terminology or ontology being 'bound to'). Each 'binding' is a URI to a target. For a binding of an id-code or an at-code, the target will be a single term, and for an ac-code, it will designate a ref-set or value set.
- value_sets: optional structure defining value-set relationships for locally defined value sets. Each value set is identified by an ac-code and has as members one or more at-codes.
- terminology_extracts: an optional structure containing extracts from external terminologies such as SNOMED CT, ICDx, or any local terminology. These extracts include the codes and preferred term rubrics, enabling the terms to be used for both display purposes. This structure is normally only used for templates, enabling small value sets for which no external reference set or subset is defined to be captured locally in the template.

Depending on whether the archetype is in differential or flat form, an instance of the `ARCHETYPE_TERMINOLOGY` class contains terms, constraints, bindings and terminology extracts that were respectively either introduced in the owning archetype, or all codes and bindings obtained by compressing an archetype lineage through inheritance. A typical instance structure of `ARCHETYPE_TERMINOLOGY` is illustrated in Figure Terminology Instance Structure.

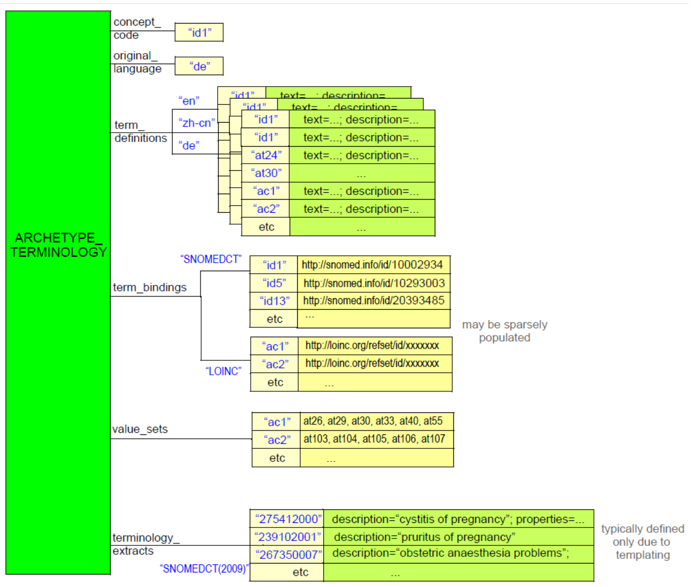

ARCHETYPE_ TERMINOLOGY
concept_ code
"id1"
original_ language
"de"
term_ definitions
"en"
"zh-cn"
"de"
"id1" text=...; description=...
"id1" text=...; description=...
"at24" text=...; description=...
"at30" ...
"ac1" text=...; description=...
"ac2" text=...; description=...
etc ...
term_bindings
"SNOMEDCT"
"id1" http://snomed.info/id/10002934
"id5" http://snomed.info/id/10293003
"id13" http://snomed.info/id/20393485
etc ...
"LOINC"
"ac1" http://loinc.org/refset/id/xxxxxxx
"ac2" http://loinc.org/refset/id/xxxxxxx
etc ...
may be sparsely populated
value_sets
"ac1" at26, at29, at30, at33, at40, at55
"ac2" at103, at104, at105, at106, at107
etc ...
terminology_ extracts
"SNOMEDCT(2009)"
"275412000" description="cystitis of pregnancy"; properties=...
"239102001" description="pruritus of pregnancy"
"267350007" description="obstetric anaesthesia problems";
etc ...
typically defined only due to templating

Figure 21. Terminology Instance Structure

7.2. Semantics

7.2.1. Specialisation Depth

Any given archetype occurs at some point in a lineage of archetypes related by specialisation, where the depth is reflected by the `specialisation_depth` function. An archetype which is not a specialisation of another has a specialisation_depth of 0. Term and constraint codes introduced in the terminology of specialised archetypes (i.e. which did not exist in the terminology of the parent archetype) are defined in a strict way, using '.' (period) markers. For example, an archetype of specialisation depth 2 will use term definition codes like the following:

- `id0.0.1` - a new term introduced in this archetype, which is not a specialisation of any previous term in any of the parent archetypes;
- `id4.0.1` - a term which specialises the 'id4' term from the top parent. An intervening '.0' is required to show that the new term is at depth 2, not depth 1;
- `id25.1.1` - a term which specialises the term 'id25.1' from the immediate parent, which itself specialises the term 'id1' from the top parent.

This systematic definition of codes enables software to use the structure of the codes to more quickly and accurately make inferences about term definitions up and down specialisation hierarchies. Constraint codes on the other hand do not follow these rules, and exist in a flat code space instead.

7.3. Class Descriptions

7.3.1. ARCHETYPE_TERMINOLOGY Class

Class	ARCHETYPE_TERMINOLOGY

Description	Local terminology of an archetype. This class defines the semantics of the terminology of an archetype.	
Attributes	**Signature**	**Meaning**
1..1	**is_differential**: `Boolean`	
1..1	**original_language**: `String`	Original language of the terminology, as set at archetype creation or parsing time; must be a code in the ISO 639-1 2 character language code-set.
1..1	**concept_code**: `String`	Term code defining the meaning of the archetype as a whole, and always used as the at-code on the root node of the archetype. Must be defined in the term_definitions property.
1..1	**term_definitions**: `Hash<Hash<ARCHETYPE_TERM, String>, String>`	Directory of term definitions as a two-level table. The outer hash keys are language codes, e.g. “en”, “de”, while the inner hash keys are term codes, e.g. “id17”, “at4”.
0..1	**term_bindings**: `Hash <Hash<Uri, String>, String>`	Directory of bindings to external terminology codes and value sets, as a two-level table. The outer hash keys are terminology ids, e.g. “SNOMED_CT”, and the inner hash keys are constraint codes, e.g. “at4”, “ac13” or paths. The indexed DV_URI objects represent references to externally defined resources, either terms, ontology concepts, or terminology subsets / ref-sets.
1..1	**parent_archetype**: `ARCHETYPE`	Archetype which owns this terminology.
0..1	**value_sets**: `Hash<VALUE_SET, String>`	Archetype-local value sets, each keyed by value-set id, i.e. an ac-code.
0..1	**terminology_extracts**: `Hash<Hash<ARCHETYPE_TERM, String>, String>`	Directory of extracts of external terminologies, as a two-level table. The outer hash keys are terminology ids, e.g. “SNOMED_CT”, while the inner hash keys are term codes or code-phrases from the relevant terminology, e.g. “10094842”.
Functions	**Signature**	**Meaning**
	specialisation_depth: `Integer`	Specialisation depth of this archetype. Unspecialised archetypes have depth 0, with each additional level of specialisation adding 1 to the specialisation_depth.
	id_codes: `List<String>`	List of all id codes in the terminology., i.e. the “id” codes in an ADL archetype, which are the node_ids on C_OBJECT descendants.

	value_codes: List<String>	List of all value term codes in the terminology, i.e. the “at” codes in an ADL archetype, which are used as possible values on terminological constrainer nodes.
	value_set_codes: List<String>	List of all value set codes in the terminology defining value sets. These correspond to the “ac” codes in an ADL archetype.
	has_language (a_lang: String): Boolean	True if language `a_lang' is present in archetype terminology.
	has_terminology (a_terminology_id: String): boolean	True if terminology `a_terminology' is present in archetype ontology.
	has_term_code (a_code: String): boolean	True if code 'a_code' defined in this terminology.
	term_definition (a_lang: String, a_code: String): ARCHETYPE_TERM ***Pre***: has_term-definition (a_lang, a_code)	Term definition for a code, in a specified language.
	term_binding (a_terminology: String, a_code: String): Uri ***Pre***: has_term_binding (a_terminology_id, a_code)	Binding of constraint corresponding to a_code in target external terminology a_terminology_id, as a string, which is usually a formal query expression.
	terminologies_available: List<String>	List of terminologies to which term or constraint bindings exist in this terminology, computed from bindings.
	terminology_extract_term (a_terminology_id: String, a_code: String): ARCHETYPE_TERM ***Pre***: has_terminology_extract (a_terminology_id) and has_terminology_extract_code (a_code)	Return an ARCHETYPE_TERM from specified terminology extract, for specified term code.
	has_terminology_extract (a_terminology_id: String): Boolean	
	languages_available: List<String>	List of languages in which terms in this terminology are available.
Invariant	***Original_language_validity***: code_set (Code_set_id_languages).has_concept_id (original_language)	
	concept_code_validity: id_codes.has (concept_code)	
	Term_bindings_validity: bindings /= void implies not bindings.is_empty	

	Parent_archetype_valid: parent_archetype.terminology = Current

7.3.2. TERMINOLOGY_RELATION Class

Class	***TERMINOLOGY_RELATION (abstract)***	
Description	Class whose instances represent any kind of 1:N relationship between a source term and 1-N target terms.	
Attributes	**Signature**	**Meaning**
1..1	**id**: `String`	Code of source term of this relation.
1..1	**members**: `List<String>`	List of target terms in this relation.

7.3.3. VALUE_SET Class

Class	**VALUE_SET**
Description	Representation of a flat value set within the archetype terminology.
Inherit	TERMINOLOGY_RELATION

7.3.4. ARCHETYPE_TERM Class

Class	**ARCHETYPE_TERM**	
Description	Representation of any coded entity (term or constraint) in the archetype ontology.	
Attributes	**Signature**	**Meaning**
1..1	**code**:	Code of this term.
1..1	**text**: `String`	Short term text, typically for display.
1..1	**description**: `String`	Full description text.
0..1	**other_items**: `Hash<String, String>`	Hash of keys and corresponding values for other items in a term, e.g. provenance. Hash of keys ("text", "description" etc) and corresponding values.

Validity Rules

The following validity rules apply to instances of this class in an archetype:

VTVSID: value-set id defined. The identifying code of a value set must be defined in the term definitions of the terminology of the current archetype.

VTVSMD: value-set members defined. The member codes of a value set must be defined in the term definitions of the terminology of the flattened form of the current archetype.

VTVSUQ: value-set members unique. The member codes of a value set must be unique within the value set.

VTSD specialisation level of codes. Term or constraint code defined in archetype terminology must be of the same specialisation level as the archetype (differential archetypes), or the same or a less specialised level (flat archetypes).

VTLC: language consistency. Languages consistent: all term codes and constraint codes exist in all languages.

VTTBK: terminology term binding key valid. Every term binding must be to either a defined archetype term ('at-code') or to a path that is valid in the flat archetype.

VTCBK: terminology constraint binding key valid. Every constraint binding must be to a defined archetype constraint code ('ac-code').

Chapter 8. Validation and Transformation Semantics

8.1. Validation

8.2. Flattening

8.3. Diff’ing

References

1. [Beale_2000] Beale T. **Archetypes: Constraint-based Domain Models for Future-proof Information Systems.** 2000. Available at http://www.openehr.org/files/resources/publications/archetypes/archetypes_beale_web_2000.pdf .
2. [Beale_2002] Beale T. **Archetypes: Constraint-based Domain Models for Future-proof Information Systems.** Eleventh OOPSLA Workshop on Behavioral Semantics: Serving the Customer (Seattle, Washington, USA, November 4, 2002). Edited by Kenneth Baclawski and Haim Kilov. Northeastern University, Boston, 2002, pp. 16-32. Available at http://www.openehr.org/files/resources/publications/archetypes/archetypes_beale_oopsla_2002.pdf .
3. [Rector] Rector A L. **Clinical terminology: why is it so hard?** Methods Inf Med. 1999 Dec;38(4-5):239-52. Available at http://www.cs.man.ac.uk/~rector/papers/Why-is-terminology-hard-single-r2.pdf .
4. [OWL] W3C. **OWL - The Web Ontology Language.** See http://www.w3.org/TR/2003/CR-owl-ref-20030818/ .
5. [Horrocks] Horrocks I, Patel-Schneider P, McGuiness D, Welty C. **OWL: a Description Logic Based Ontology Language for the Semantic Web** See http://www.cs.ox.ac.uk/ian.horrocks/Publications/download/2003/HPMW07.pdf .
6. [OCL] **The Object Constraint Language 2.0.** Object Management Group (OMG). Available at http://www.omg.org/cgi-bin/doc?ptc/2003-10-14 .