

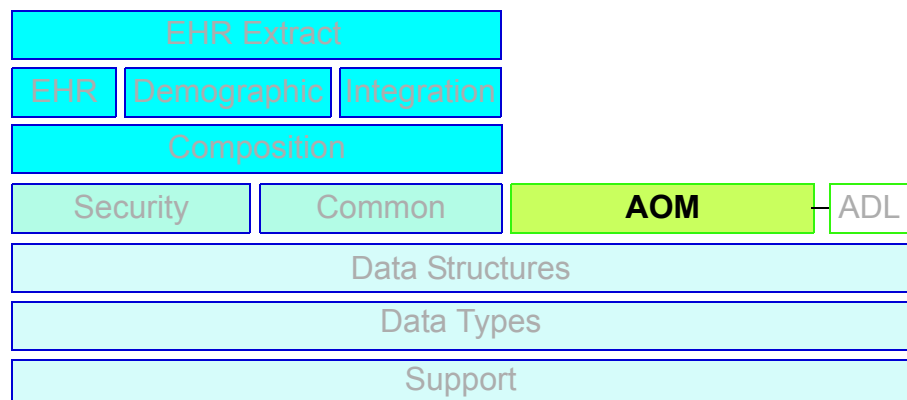


The *openEHR* Archetype Model

Archetype Object Model

<i>Issuer:</i> openEHR Specification Program		
<i>Revision:</i> 2.1.10	<i>Pages:</i> 101	<i>Date of issue:</i> 18 Jul 2014
<i>Status:</i> TRIAL		

Keywords: EHR, ADL, health records, archetypes, constraints



© 2004- The *openEHR* Foundation.

The *openEHR* Foundation is an independent, non-profit community, facilitating the sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Affiliates Australia, Brazil, Japan, New Zealand, Portugal, Sweden

Licence  Creative Commons Attribution-NoDerivs 3.0 Unported.
creativecommons.org/licenses/by-nd/3.0/

Support **Issue tracker:** www.openehr.org/issues/browse/SPECPR
Web: www.openEHR.org

Amendment Record

Issue	Details	Raiser	Completed
RELEASE 1.1 candidate			
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. <i>uid</i> to <i>urn</i> :URN.	I McNicoll, S Garde, T Beale	04 Jun 2014
2.1.8	Rename ARCHETYPE. <i>commit_number</i> to <i>build_count</i> .	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 <i>ontology</i> to <i>terminology</i> 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 Aug 2013
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. <ul style="list-style-type: none"> move C_PRIMITIVE.<i>assumed_value</i> to attribute slot in UML rename C_DEFINED_OBJECT.<i>default_value</i> function to <i>prototype_value</i> correct <i>assumed_value</i> 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.<i>is_template</i> attribute. add ARCHETYPE.<i>is_component</i> attribute. allow computed as well as stored attributes. make ONTOLOGY.<i>terminologies_available</i> computed. 	T Beale	10 Dec 2009

Issue	Details	Raiser	Completed
2.0.9	<p>SPEC-263. Change Date, Time etc classes in AOM to ISO8601_DATE, ISO8601_TIME etc from Support IM.</p> <p>SPEC-296. Convert Interval<Integer> to MULTIPLICITY_INTERVAL to simplify specification and implementation.</p> <p>SPEC-300. Archetype slot regular expressions should cover whole identifier. Added C_STRING.is_pattern.</p> <p>SPEC-303. Make <i>existence</i>, <i>occurrences</i> and <i>cardinality</i> optional in AOM.</p> <p>SPEC-308. Add validity rules to ARCHETYPE_TERMINOLOGY.</p> <p>SPEC-309. ARCHETYPE_CONSTRAINT adjustments.</p> <p>SPEC-178. Add template object model to AM.</p> <ul style="list-style-type: none"> Add <i>is_exhaustive</i> attribute to ARCHETYPE_SLOT. Add <i>is_template</i> attribute to ARCHETYPE. Add <i>terminology_extracts</i> to ARCHETYPE_TERMINOLOGY. 	<p>T Beale</p> <p>T Beale</p> <p>A Flinton</p> <p>S Heard</p> <p>T Beale</p> <p>T Beale</p> <p>T Beale</p>	20 Jul 2009
RELEASE 1.0.2			
2.0.2	<p>SPEC-257. Correct minor typos and clarify text. Correct reversed definitions of <i>is_bag</i> and <i>is_set</i> in CARDINALITY class.</p> <p>SPEC-251. Allow both pattern and interval constraint on Duration in Archetypes. Add <i>pattern</i> attribute to C_DURATION class.</p>	<p>C Ma,</p> <p>R Chen,</p> <p>T Cook</p> <p>S Heard</p>	20 Nov 2008
RELEASE 1.0.1			
2.0.1	<p>CR-000200. Correct Release 1.0 typographical errors. Table for missed class ASSERTION_VARIABLE added. Assumed_value assertions corrected; <i>standard_representation</i> function corrected. Added missed <i>adl_version</i>, <i>concept</i> rename from CR-000153.</p> <p>CR-000216: Allow mixture of W, D etc in ISO8601 Duration (deviation from standard).</p> <p>CR-000219: Use constants instead of literals to refer to terminology in RM.</p> <p>CR-000232. Relax validity invariant on CONSTRAINT_REF.</p> <p>CR-000233: Define semantics for <i>occurrences</i> on ARCHETYPE_INTERNAL_REF.</p> <p>CR-000234: Correct functional semantics of AOM constraint model package.</p> <p>CR-000245: Allow term bindings to paths in archetypes.</p>	<p>D Lloyd,</p> <p>P Pazos,</p> <p>R Chen,</p> <p>C Ma</p> <p>S Heard</p> <p>R Chen</p> <p>R Chen</p> <p>K Atalag</p> <p>T Beale</p> <p>S Heard</p>	20 Mar 2007
RELEASE 1.0			
2.0	<p>CR-000153. Synchronise ADL and AOM attribute naming.</p> <p>CR-000178. Add Template Object Model to AM. Text changes only.</p> <p>CR-000167. Add AUTHORED_RESOURCE class. Remove description package to resource package in Common IM.</p>	<p>T Beale</p> <p>T Beale</p> <p>T Beale</p>	10 Nov 2005
RELEASE 0.96			

Issue	Details	Raiser	Completed
0.6	<p>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.</p> <p>CR-000142. Update ADL grammar to support assumed values. Changed C_PRIMITIVE and C_DOMAIN_TYPE.</p> <p>CR-000146: Alterations to am.archetype.description from CEN MetaKnow</p> <p>CR-000138. Archetype-level assertions.</p> <p>CR-000157. Fix names of OPERATOR_KIND class attributes</p>	<p>D Lloyd</p> <p>S Heard, T Beale D Kalra</p> <p>T Beale T Beale</p>	20 Jun 2005
RELEASE 0.95			
0.5.1	Corrected documentation error - return type of ARCHETYPE_CONSTRAINT. <i>has_path</i> ; add optionality markers to Primitive types UML diagram. Removed erroneous aggregation marker from ARCHETYPE_ONTOLOGY. <i>parent_archetype</i> and ARCHETYPE_DESCRIPTION. <i>parent_archetype</i> .	D Lloyd	20 Jan 2005
0.5	<p>CR-000110. Update ADL document and create AOM document. Includes detailed input and review from:</p> <ul style="list-style-type: none"> - DSTC - CHIME, Uuniversity College London - Ocean Informatics <p>Initial Writing. Taken from ADL document 1.2draft B.</p>	<p>T Beale</p> <p>A Goodchild Z Tun T Austin D Kalra N Lea D Lloyd S Heard T Beale</p>	10 Nov 2004

Trademarks

Microsoft is a trademark of the Microsoft Corporation

Acknowledgements

The work reported in this document was funded by:

- Ocean Informatics;
- 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
- Dipak Kalra MD, PhD, Professor Health Informatics, UCL, president Eurorec
- Shinji Kobayashi PhD, Kyoto University EHR research unit
- Bostjan Lah, Architect, Marand, Slovenia
- 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, Linköping University, Sweden
- Alessandro Torrisi, Code24, The Netherlands.

Table of Contents

1	Introduction.....	9
1.1	Purpose	9
1.2	Related Documents.....	9
1.3	Nomenclature	9
1.4	Status	9
1.5	Tools	10
1.6	Changes from Previous Versions.....	10
1.6.1	Release 1.4 to 1.5 (Document version 2.0 to 2.1.x)	10
1.6.2	Release 0.6 to 1.4	10
2	Background	11
2.1	Architectural Context	11
2.2	Basic Semantics.....	11
2.2.1	Archetype Relationships	12
2.3	The Development Environment	13
2.3.1	Model / Syntax Relationship	13
2.3.2	The Development Process	13
2.3.3	Compilation.....	14
2.3.4	Optimisations	15
3	Model Overview	17
3.1	Package Structure	17
4	Support Package	18
4.1	Overview	18
4.1.1	Utility Classes	18
4.1.2	ADL_15_CODE_DEFINITIONS Class	18
4.2	Class Definitions	18
5	The Archetype Package.....	22
5.1	Overview	22
5.1.1	Archetype Identification.....	22
5.1.2	Reference Model Release.....	24
5.1.3	Descriptive Meta-data	24
5.1.4	Common Structural Parts	24
5.1.5	Structural Variants	26
5.2	Class Descriptions	30
5.2.1	ARCHETYPE Class.....	30
5.2.2	ARCHETYPE_HRID Class	32
5.2.3	DIFFERENTIAL_ARCHETYPE Class	34
5.2.4	FLAT_ARCHETYPE Class	34
5.3	Validity Rules	36
6	Constraint Model Package.....	38
6.1	Overview	38
6.2	Semantics.....	41
6.2.1	All Node Types.....	41
6.2.2	Attribute Node Types	41
6.2.3	Object Node Types	42

6.3	Second Order Constraints	46
6.3.1	Tuple Constraints	47
6.3.2	Group Constraint	48
6.3.3	Assertions	49
6.4	C_OBJECT Substitutions	49
6.5	Class Definitions	50
6.5.12	C_BOOLEAN Class	69
6.5.13	C_STRING Class	69
6.5.14	C_ORDERED Class	70
6.5.15	C_INTEGER Class	70
6.5.16	C_REAL Class	71
6.5.17	C_TEMPORAL Class	71
6.5.18	C_DATE Class	71
6.5.19	C_TIME Class	72
6.5.20	C_DATE_TIME Class	72
6.5.21	C_DURATION Class	73
6.5.22	C_TERMINOLOGY_CODE Class	74
6.5.23	TERMINOLOGY_CODE Class	75
7	The Assertion Package	77
7.1	Overview	77
7.2	Semantics	78
7.3	Class Descriptions	78
7.3.1	RULE_STATEMENT Class	78
7.3.2	ASSERTION Class	78
7.3.3	VARIABLE_DECLARATION Class	79
7.3.4	EXPR_VARIABLE Class	79
7.3.5	BUILTIN_VARIABLE Class	80
7.3.6	QUERY_VARIABLE Class	80
7.3.7	EXPR_ITEM Class	81
7.3.8	EXPR_LEAF Class	81
7.3.9	EXPR_CONSTANT Class	82
7.3.10	EXPR_CONSTRAINT Class	82
7.3.11	EXPR_ARCHETYPE_ID_CONSTRAINT Class	82
7.3.12	EXPR_MODEL_REF Class	83
7.3.13	EXPR_VARIABLE_REF Class	83
7.3.14	EXPR_OPERATOR Class	84
7.3.15	EXPR_UNARY_OPERATOR Class	84
7.3.16	EXPR_BINARY_OPERATOR Class	84
7.3.17	OPERATOR_KIND Class	86
8	Terminology Package	88
8.1	Overview	88
8.2	Semantics	89
8.2.1	Specialisation Depth	89
8.3	Class Descriptions	90
8.3.1	ARCHETYPE_TERMINOLOGY Class	90
8.3.2	ARCHETYPE_TERM Class	93
8.3.3	VALUE_SET Class	94

Algorithms 95

1 Introduction

1.1 Purpose

This document contains the definitive formal statement of archetype and template semantics, in the form of an object model for archetypes. 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) [4], XML-instance and so on. As a specification, it can be treated as an API for archetypes within software tools and components.

It is recommended that the *openEHR* ADL document [4] 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 is nominally ‘1.5’ based on the version of the ADL specification, even though this specification document is in version 2.1.x. Thus ‘AOM 1.5’ refers to the version of AOM compatible with ‘ADL 1.5’.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The [openEHR Architecture Overview](#)

Related documents include:

- The *openEHR* [Archetype Definition Language \(ADL\)](#)
- The *openEHR* 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 1.5 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 document is under development, and is published as a proposal for input to standards processes and implementation works.

The development version of this document can be found at <http://www.openehr.org/releases/trunk/architecture/am/aom1.5.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 1.4 to 1.5 (Document version 2.0 to 2.1.x)

The changes in version 2.1 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 definition, rather than including a copy of unchanged elements. Doing the latter is clearly unsustainable in terms of change management. The 2.0 model already supported differential representation, but somewhat inconveniently.

The changes are as follows.

- Introduction of **new archetype 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;
- **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 and renaming to ‘terminology’;
- **Expression of all external term bindings as URIs** following IHTSDO format;
- Introduction of **generic ‘tuple’ second order 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;
- The name of the *invariant* section has been changed to *rules*, to better reflect its purpose.

1.6.2 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*.

2 Background

2.1 Architectural Context

Archetypes form the second layer of the *openEHR* semantic architecture. They provide a way of creating models of domain content, expressed in terms of constraints on a reference model. Archetype paths provide the basis of querying in *openEHR* as well as bindings to terminology.

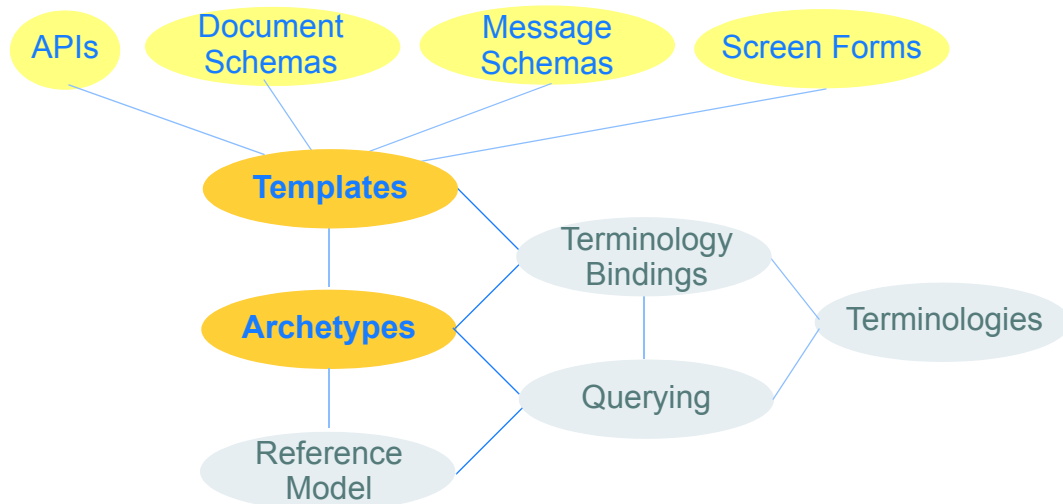


FIGURE 1 The *openEHR* Semantic Architecture

The semantics of archetypes and templates are defined by the following specifications:

- the *openEHR* Archetype Object Model (AOM);
- the [Archetype Definition Language \(ADL\)](#).

The AOM is the definitive formal expression of archetype semantics, and is independent of any particular syntax. **The main purpose of the AOM specification is to specify to developers how to build archetype tools and also EHR components that use archetypes.**

The Archetype Definition Language (ADL) is a formal abstract syntax for archetypes, and can be used to provide a default serial expression of archetypes. It is the **primary document for understanding the semantics of archetypes.**

The semantics defined in the AOM are used to express the object structures of source archetypes and flattened archetypes. Since in AOM 1.5 a template is just a kind of archetype, the AOM also describes the semantics of templates as well. The two source forms are authored by users using tools, while the two flat forms are generated by tools. The rules for how to use the AOM for each of these forms is described in details in this specification.

2.2 Basic Semantics

Archetypes are topic- or theme-based models of domain content, expressed in terms of constraints on a reference information model. Since each archetype constitutes an encapsulation of a set of data points pertaining to a topic, it is of a manageable, limited size, and has a clear boundary. For example an 'Apgar result' archetype of the *openEHR* reference model class `OBSERVATION` contains the data points relevant to Apgar score of a newborn, while a 'blood pressure measurement' archetype con-

tains data points relevant to the result and measurement of blood pressure. Archetypes are assembled by templates to form structures used in computational systems, such as document definitions, message definitions and so on.

2.2.1 Archetype Relationships

A ‘system’ of archetypes is a collection of archetypes covering all or part of a domain, such as clinical medicine. Apart from versioning, two kinds of relationship can exist between archetypes in the system: specialisation and composition. The specialisation relationship in particular affects the parsing and validation of archetypes in the system.

Archetype Specialisation

An archetype can be specialised in a descendant archetype in a similar way to a subclass in an object-oriented programming language. Specialised archetypes are, like classes, expressed in a *differential* form with respect to the parent archetype. This is a necessary pre-requisite to sustainable management of specialised archetypes. An archetype is a specialisation of another archetype if it mentions that archetype as its parent, and only makes changes to its definition such that its constraints are ‘narrower’ than those of the parent. Note that this can include a specialised archetype defining constraints on a part of the reference model not constrained at all by parent archetypes - since this is still ‘narrowing’ or constraining. The chain of archetypes from a specialised archetype back through all its parents to the ultimate parent is known as an *archetype lineage*. For a non-specialised (i.e. top-level) archetype, the lineage is just itself.

In order for specialised archetypes to be *used*, the differential form used for authoring has to be *flattened* through the archetype lineage to create *flat-form archetypes*, i.e. the standalone equivalent of a given archetype, as if it had been constructed on its own. A flattened archetype is expressed in the same serial and object form as a differential form archetype, although there are some slight differences in the semantics, other than the use of differential paths.

Any data created via the use of an archetype conforms to the flat form of the archetype, and to the flat form of every archetype up the lineage.

The semantics of specialisation are described in detail in the *openEHR* ADL specification.

Archetype Composition

In the interests of re-use, archetypes can be composed to form larger structures semantically equivalent to a single large archetype. Composition allows two things to occur: for archetypes to be defined according to natural ‘levels’ or encapsulations of information, and for the re-use of smaller archetypes by higher-level archetypes. There are two mechanisms for expressing composition: direct reference, and archetype *slots* which are defined in terms of constraints. The latter, unlike an object model, allows an archetype to have a composition relationship with any number of archetypes matching some constraint pattern. Depending on what archetypes are available within the system, the archetypes matched may vary.

Templates

In practical systems, archetypes are assembled into larger usable structures by the use of *openEHR* templates. A template is expressed in the same source form as a specialised archetype, although making use of the slot-filling mechanism. It is processed against an archetype library to product an *operational template*. The latter is like a large flat-form archetype, and is the form used for runtime validation, and also for the generation of all computational artefacts derived from templates. Semantically, templates perform three functions: aggregating multiple archetypes, removing elements not needed for the use case of the template, and narrowing some existing constraints, in the same way as

specialised archetypes. The effect is to re-use needed elements from the archetype library, arranged in a way that corresponds directly to the use case at hand.

The AOM does not distinguish between archetypes, specialised archetypes or templates other than by use an an artefact type classifier. All other differences in how archetypes and templates work are implemented in tools that may prevent or allow certain operations depending on whether the artefact being worked on is an archetype or template.

2.3 The Development Environment

2.3.1 Model / Syntax Relationship

The AOM can be considered as the model of an in-memory archetype or a template, or equivalently, the syntax tree for any syntax form of the same. The abstract syntax form of an archetype is ADL, but an archetype may just as easily be parsed from and serialised to XML, JSON or any other format. The in-memory archetype representation may also be created by calls to a suitable AOM construction API, from an archetype or template editing tool. These relationships, and the relation between each form and its specification are shown in FIGURE 2.

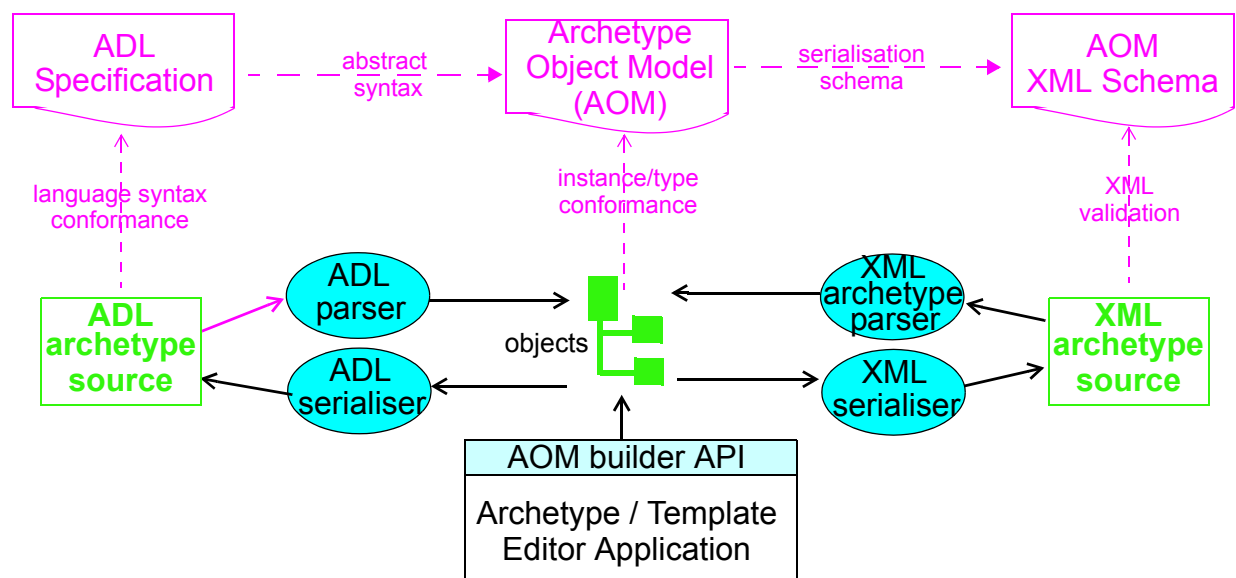


FIGURE 2 Relationship of archetype in-memory and syntax forms

The existence of source and flat form archetypes and templates, potentially in multiple serialised formats may initially appear confusing, although any given environment tends to use a single serialised form. FIGURE 3 illustrates all possible archetype and template artefact types, including file types, and shows which specifications they are defined by.

2.3.2 The Development Process

Archetypes and templates are authored and transformed according to a number of steps very similar to class definitions within an object-oriented programming environment. The activities in the process are as follows:

- *archetype authoring*: creates source-form archetypes, expressed in AOM objects;
- *archetype validation*: creates flattened archetypes;

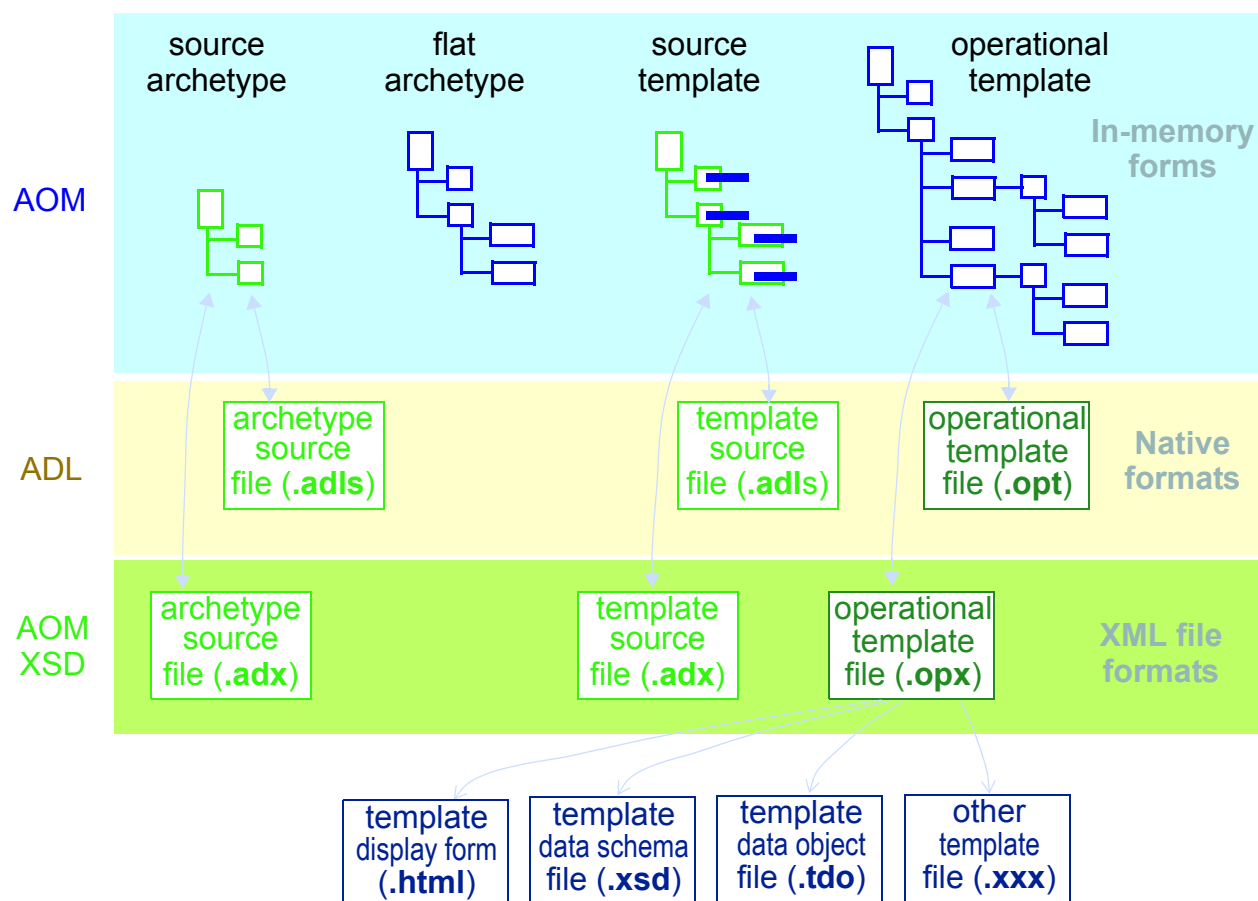


FIGURE 3 Relationship of computational artefacts and specifications

- *template authoring*: creates source-form templates that reference archetypes; also expressed as AOM objects;
- *operational template generation*: creates fully flattened archetype-based templates.

The tool chain for the process is illustrated in FIGURE 4. From a business point of view, template authoring is the starting point. A template references one or more archetypes, so its compilation (parsing, validation, flattening) involves both the template source and the validated, flattened forms of the referenced archetypes. With these as input, a template flattener can generate the final output, an operational template.

2.3.3 Compilation

A tool that parses, validates, flattens and serialises a library of archetypes is called a compiler. Due to archetype specialisation, *archetype specialisation ineages* rather than just single archetypes are processed - i.e. specialised archetypes can only be compiled in conjunction with their specialisation parents up to the top level. This is exactly how object-oriented programming environments work. For any given lineage, compilation proceeds from the top-level archetype downward. Each archetype is validated, and if it passes, flattened with the parent in the chain. This continues until the archetype originally being compiled is reached. In the many cases of archetypes with no specialisations, compilation involves the one archetype only.

FIGURE 5 illustrates the object structures for an archetype lineage as created by a compilation process, with the elements corresponding to the top-level archetype bolded. Differential input file(s) are

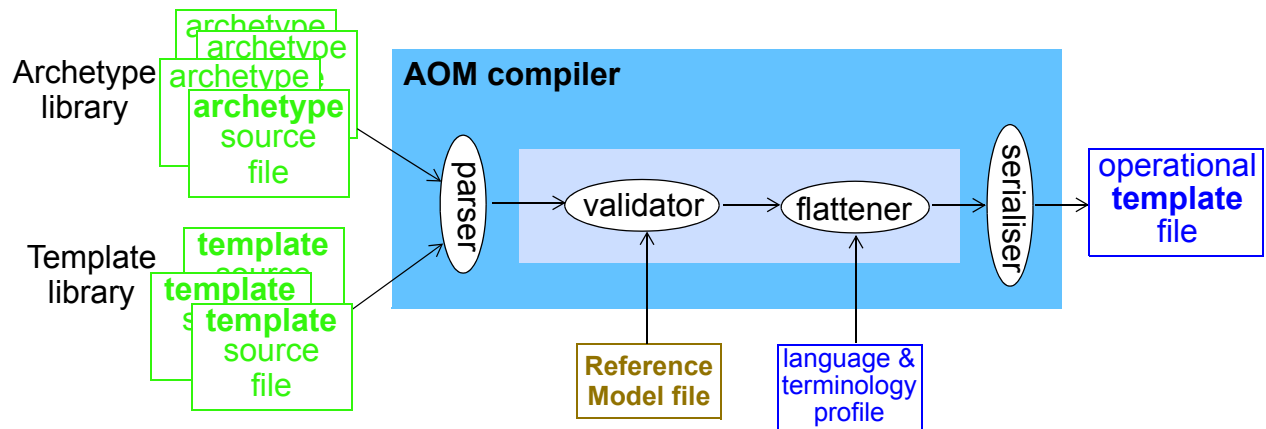


FIGURE 4 Archetype / template tool chain

converted by the parser into differential object parse trees, shown at the right of the figure. The same structures would be created by an editor application.

The differential in-memory representation is validated by the semantic checker, which verifies numerous things, such as that term codes referenced in the definition section are defined in the terminology section. It can also validate the classes and attributes mentioned in the archetype against a specification for the relevant reference model¹.

The results of the compilation process can be seen in the archetype visualisations in the *openEHR* ADL Workbench².

2.3.4 Optimisations

There is a subtlety in dealing with syntax and in-memory forms of archetypes and templates which becomes important in *openEHR* system design. Artefacts authored by whatever means, including by users with a tool (which may be as simple as a text editor), should always be considered ‘suspect’ until proven otherwise by reliable validation. This is true regardless of the original syntax - ADL, XML or something else. Once validated however, the flat form can be reserialised both in a format suitable for editor tools to use (ADL, XML, ...), and also in a format that can be regarded as a reliable pure object serialisation of the in-memory structure. The latter form is often XML-based, but can be any object representation form, such as JSON, the *openEHR* ODIN (aka dADL) syntax, YAML, a binary form, or a database structure. It will not be an abstract syntax form such as ADL, since there is an unavoidable semantic transformation required between the abstract syntax and object form.

The utility of this pure object serialisation is that it can be used as *persistence* of the validated artefact, to be converted to in-memory form using only generic object deserialisation, rather than the typical multi-pass compiler/validator that needs to be used for parsing an artefact of unreliable / unknown origin. This allows such validated artefacts to be used in both design environments and more importantly, runtime systems with no danger of compilation errors. It is the same principle used in creating .jar files from Java source code, and .Net assemblies from C# source code.

1. An ODIN expression of the *openEHR* reference model is available for this purpose.

2. See http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/apps/doc/adl_workbench_help.htm

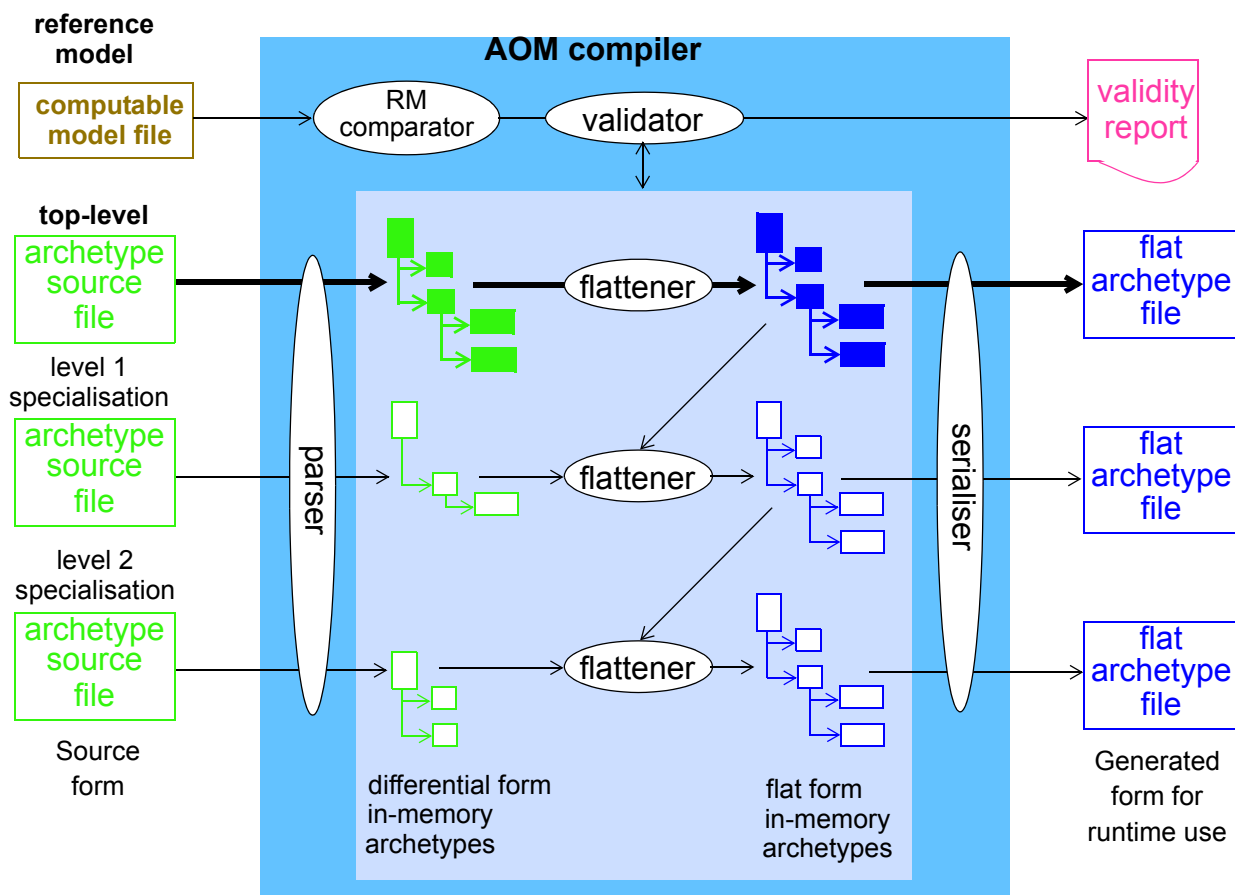


FIGURE 5 Computational model of archetype compilation

Within *openEHR* environments, managing the authoring and persisted forms of archetypes is achieved using various mechanisms including digital signing, which are described in the *openEHR* Distributed Development Model document.

3 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. It is independent of any particular serialised expression of an archetype, such as ADL or XML, and can therefore be used with any kind of parser.

It is dependent on the *openEHR* Support model (assumed types and identifiers), as small number of the *openEHR* Data types IM, and the `AUTHORED_RESOURCE` classes from the *openEHR* Common IM.

3.1 Package Structure

The *openEHR* Archetype Object Model is defined as the package `am.archetype`, as illustrated in FIGURE 6. It is shown in the context of the *openEHR* `am.archetype` packages.

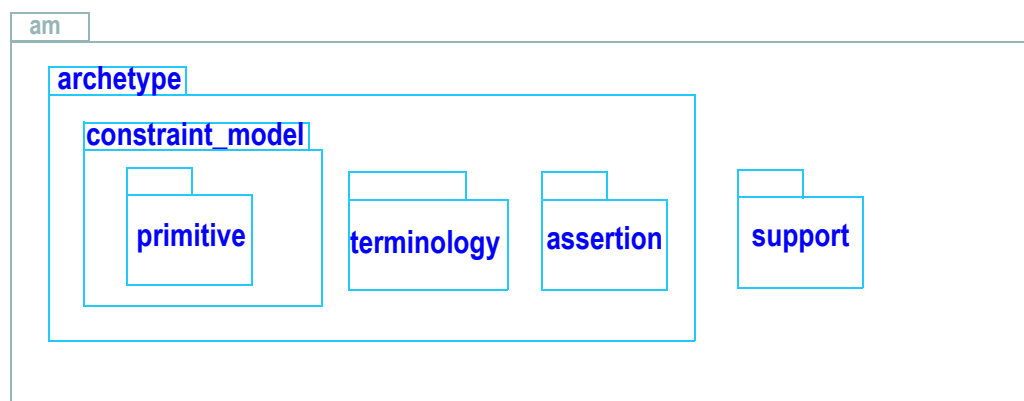


FIGURE 6 openehr.am.archetype Package

4 Support Package

4.1 Overview

The `am.support` package contains a number of classes providing definitions used elsewhere in the archetype object model, and is illustrated below.

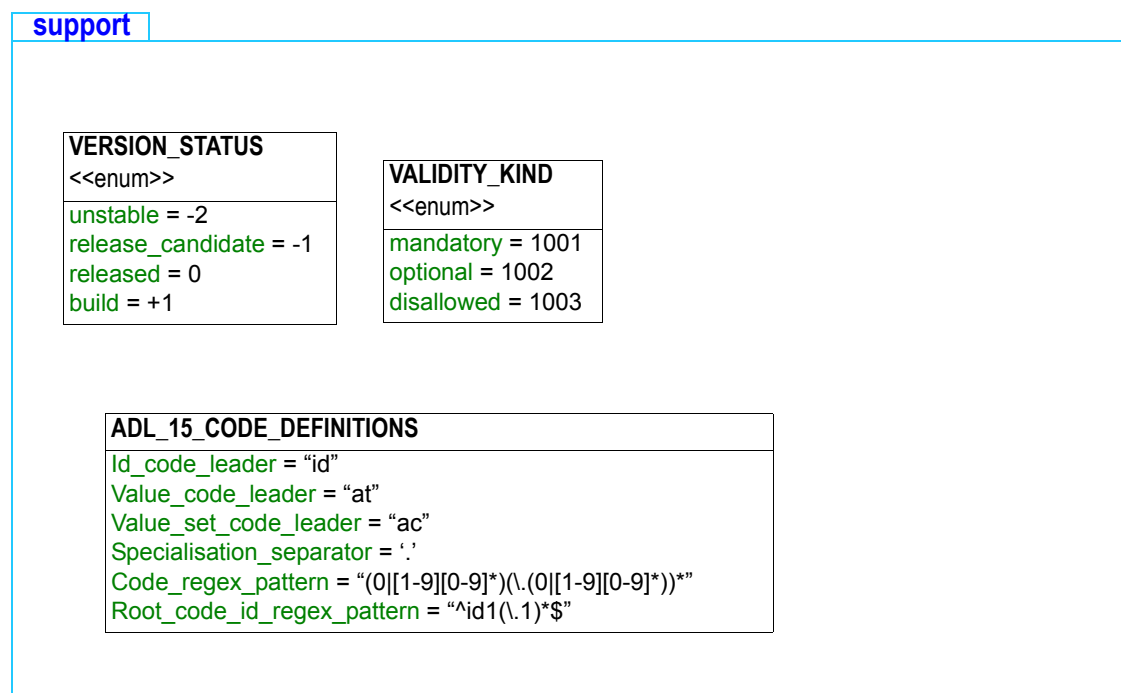


FIGURE 7 openehr.am.support package

4.1.1 Utility Classes

The enumeration class, `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`.

4.1.2 ADL_15_CODE_DEFINITIONS Class

This class provides formal definitions for a number of constants relating to the archetype internal coding system.

4.2 Class Definitions

4.2.1 VERSION_STATUS Class

CLASS	VERSION_STATUS
Purpose	Status of this version, as one of a number of possible values: uncontrolled, pre-release, release, build.

CLASS	VERSION_STATUS	
Attributes	Signature	Meaning
	unstable: Integer = -2	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+uB" e.g. "2.0.1+u154".
	release_candidate: Integer = -1	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-rcB" e.g. "2.0.1-rc27".
	released: Integer = 0	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: Integer = 1	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".
Invariant		

4.2.2 VALIDITY_KIND Class

CLASS	VALIDITY_KIND	
Purpose	An enumeration of three values which may commonly occur in constraint models.	
Use	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
1	mandatory: Integer = 1001	Constant to indicate mandatory presence of something
1	optional: Integer = 1002	Constant to indicate optional presence of something
1	disallowed: Integer = 1003	Constant to indicate disallowed presence of something
Functions	Signature	Meaning

CLASS	VALIDITY_KIND	
	valid_validity (n: Integer) : Boolean <i>ensure</i> n >= mandatory and n <= disallowed	Function to test validity values.
Invariant		

4.2.3 ADL_15_CODE_DEFINITIONS Class

CLASS	ADL_15_CODE_DEFINITIONS	
Purpose	Definitions relating to the internal code system of archetypes.	
Use	Provides constants required in other classes.	
Attributes	Signature	Meaning
1	Id_code_leader : String = "id"	String leader of 'identifier' codes, i.e. codes used to identify archetype nodes.
1	Value_code_leader : String = "at"	String leader of 'value' codes, i.e. codes used to identify codes values, including value set members.
1	Value_set_code_leader : String = "ac"	String leader of 'value set' codes, i.e. codes used to identify value sets.
1	Specialisation_separator : Character = '.'	Character used to separate numeric parts of codes belonging to different specialisation levels.
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	Root_code_id_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..
Functions	Signature	Meaning
	valid_validity (n: Integer) : Boolean <i>ensure</i> n >= mandatory and n <= disallowed	Function to test validity values.

CLASS	ADL_15_CODE_DEFINITIONS
Invariant	

5 The Archetype Package

5.1 Overview

The top-level model of archetypes and templates (all variant forms) is illustrated in FIGURE 8. The model defines a standard structural representation of an archetype, typically deserialised from either the *openEHR* ADL format, or an ‘object graph’ serialisation in a format such as ODIN, XML, JSON or YAML. The class `ARCHETYPE` is a descendant of the `AUTHORED_RESOURCE` class, which provides a standardised model of descriptive meta-data, language information, annotations and revision history. The `ARCHETYPE` class adds a number of identifying attributes and flags, along with the substantive parts of the definition, namely *definition*, optional *rules* part, and *terminology*.

5.1.1 Archetype Identification

All archetype variants have a number of identifying properties defined by the `ARCHETYPE` and associated `ARCHETYPE_HRID` classes. The *archetype_id* attribute defines the mandatory multi-axial identifier of an archetype, while the optional *uid* attribute supports an ISO Oid or GUID style identifier. This will probably become mandatory in the future, in order to properly support lifecycle management. Archetypes created new in tools should be given a GUID as a default uid.

The class `ARCHETYPE_HRID` defines an atomised representation of the Archetype *archetype_id*, enabling variant forms to be used as needed. Its various parts can be understood from the following diagram, which also shows the computed *interface_id* and *physical_id* in various forms. Note that for specialised archetypes, the *parent_archetype_id* is simply a string reference to an archetype, and should normally only use the ‘interface’ variant of the id, i.e. down to the major version only. A full explanation of the identification system is given in the openEHR Knowledge Artefact Identification specification.

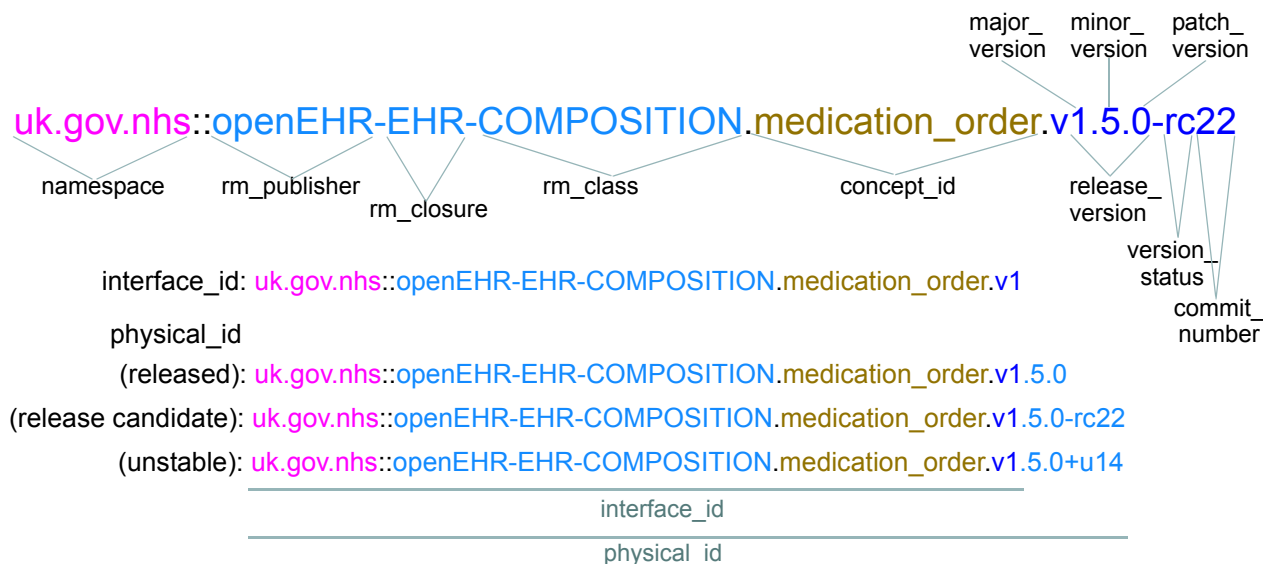


FIGURE 9 ARCHETYPE_HRID structure

The *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

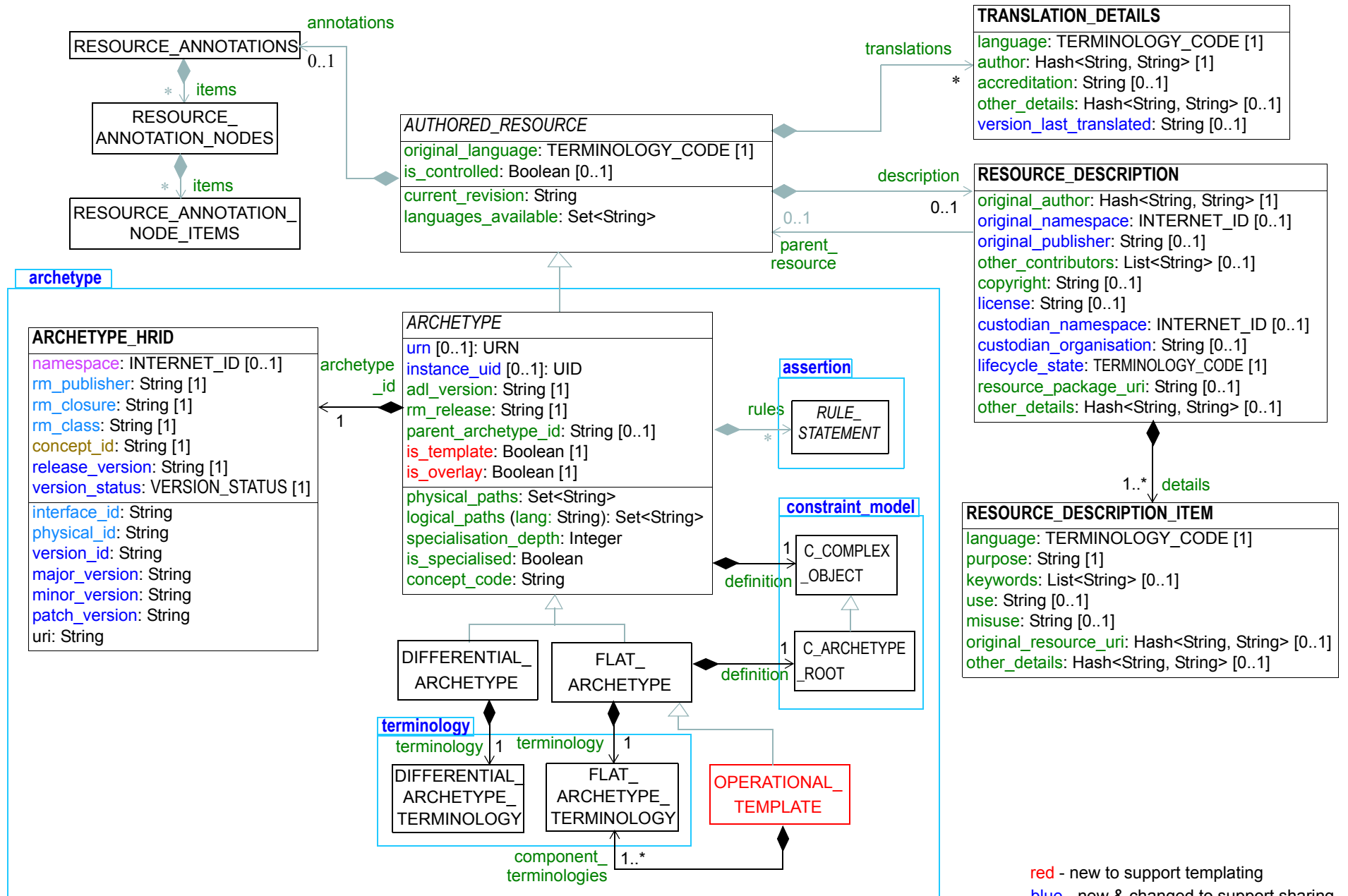


FIGURE 8 openehr.am.archetype Package

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.

The *instance_uid* attribute is used by tools to identify each changed revision of an artefact, regardless of how small the change is. A new value is assigned for every change saved in a tool environment. If an archetype is uploaded to a model management environment containing an *instance_uid*, and no changes are made during the process of committal to the repository, no change is required for this field. On the other hand, if it is not set on commit, it should be populated. For every change made to the 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.

5.1.2 Reference Model Release

The *rm_release* property of the `ARCHETYPE` class designates the release id of the reference model on which the archetype is based, in the archetype's *current* version. This should be in the same sem-ver.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 versions other than the named one, since most archetypes can easily conform to more than one. More minimal archetypes are likely to technically conform to more old and future releases than more complex archetypes.

5.1.3 Descriptive Meta-data

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 two distinct part of the meta-data: core and description details. The core consists of items such as author names, contributors, *lifecycle_state*, etc that are not translated, while the description/details part consists of natural language items that are translated to all of the languages of the archetype. The core meta-da includes the translation information. The following screenshots provide an impression of typical archetype meta-data.

Most meta-data items are self-explanatory, however the following deserve some explanation.

Lifecycle_state

The *description.lifecycle_state* is an important property of an archetype, which is used to record its state in a defined lifecycle, illustrated in FIGURE 12. The lifecycle 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', 'initial', 'unstable', etc.

Custodian_namespace and Custodian_organisation

These two properties provide a formal namespace, and a human-readable organisation identifier corresponding to the current custodian of the artefact. They enable the user of the artefact to determine who is the definitive maintainer and publisher.

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.

5.1.4 Common Structural Parts

The archetype *definition* is the main definitional part of an archetype and is an instance of a `C_COMPLEX_OBJECT` (or its descendant `C_ARCHETYPE_ROOT`, in the case of a template). This means that

Authoring		Details
Lifecycle state: unmanaged		
Authorship		
Original Author		Contributors
name	Sam Heard	Sebastian Garde, Ocean Informatics, Germany
organisation	Ocean Informatics	Anneke Goossen, Results 4 Care, Netherlands
email	sam.heard@oceaninformatics.com	Sam Heard, Ocean Informatics, Australia
date	22/03/2006	Karsten Heusser, Hannover Medical School, Germany
		Omer Hotomaroglu, Turkey
		Evelyn Hovenga, EJS Consulting, Australia
		Derek Hoy, United Kingdom
		Pieter Hummel, Medisch Centrum Alkmaar, Netherlands
		Eugene Igras, IRIS Systems, Inc., Canada
Languages		
Original language:	en	Translations de
Translator		Translator Other Det
name	Sebastian Garde, Jasmin Buck	
organisation	Ocean Informatics, University of Heidelberg	
Translator Accreditation		
Copyright:	© openEHR Foundation	

FIGURE 10 Authoring meta-data

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 on page 88.

An archetype may include one or more *rules*. Rules are statements expressed in a subset of predicate logic, which can be used to state constraints on multiple parts of an object. They are not needed to constrain single attributes or objects (since this can be done with an appropriate `C_ATTRIBUTE` or `C_OBJECT`), but are necessary for constraints referring to more than one attribute, such as a constraint that ‘systolic pressure should be \geq 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.

Authoring		Details	Keyword
Archetype Details			
Purpose:	To record the systemic arterial blood pressure of an individual.		observations measurement bp vital signs mean arterial pressure pulse pressure systolic diastolic RR NIBP
Use:	<p>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.</p> <p>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.</p> <p>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).</p>		
Misuse:	<p>Not to be used for intravenous pressure.</p> <p>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.</p> <p>Use OBSERVATION.intravascular_pressure and related specialisations in both of these situations.</p>		
Resources			
Package			
Original Resources			

FIGURE 11 Translatable meta-data

5.1.5 Structural Variants

The model in FIGURE 8 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 13 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.

For either immediate descendant of the `ARCHETYPE` class, if the property `is_template` = True, the structure is a template.

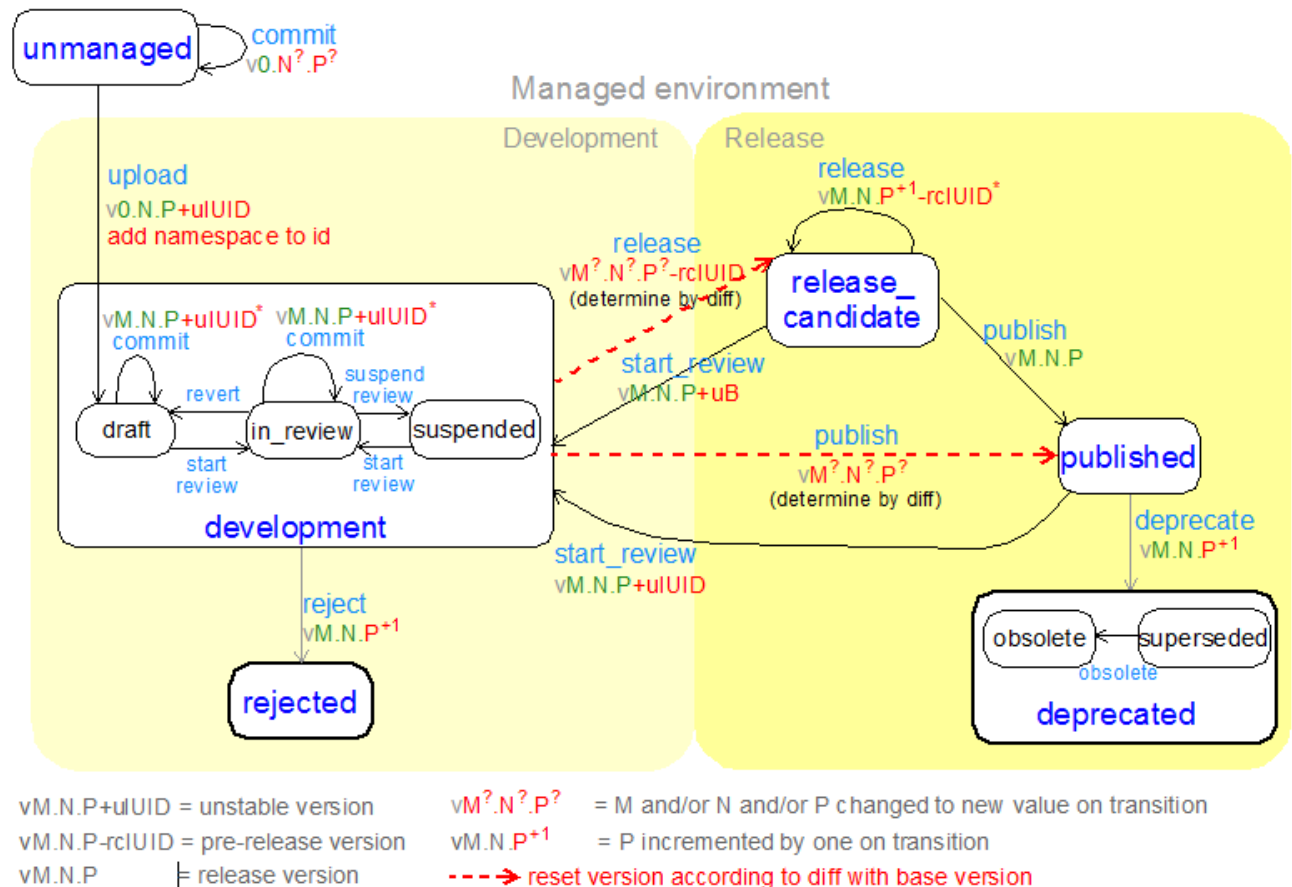


FIGURE 12 Development Lifecycle and Versioning

FIGURE 14 illustrates the structure of a *source template*, i.e. instances of `DIFFERENTIAL_ARCHETYPE` class with `is_template = True`. 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 FIGURE 14 is the *template overlay*, i.e. `DIFFERENTIAL_ARCHETYPE` instances with `is_template`, `is_overlay` and `is_specialised` all set to `True`. 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 program-

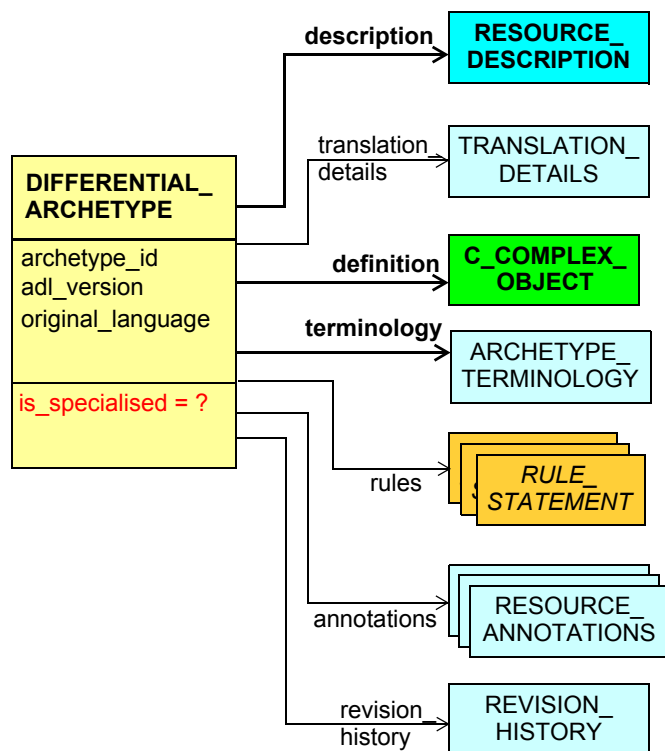


FIGURE 13 Source archetype instance structure

ming languages. (There is no technical reason why overlay *archetypes*, i.e. `is_template = False`, could not also exist, but as yet there seems no obvious practical use).

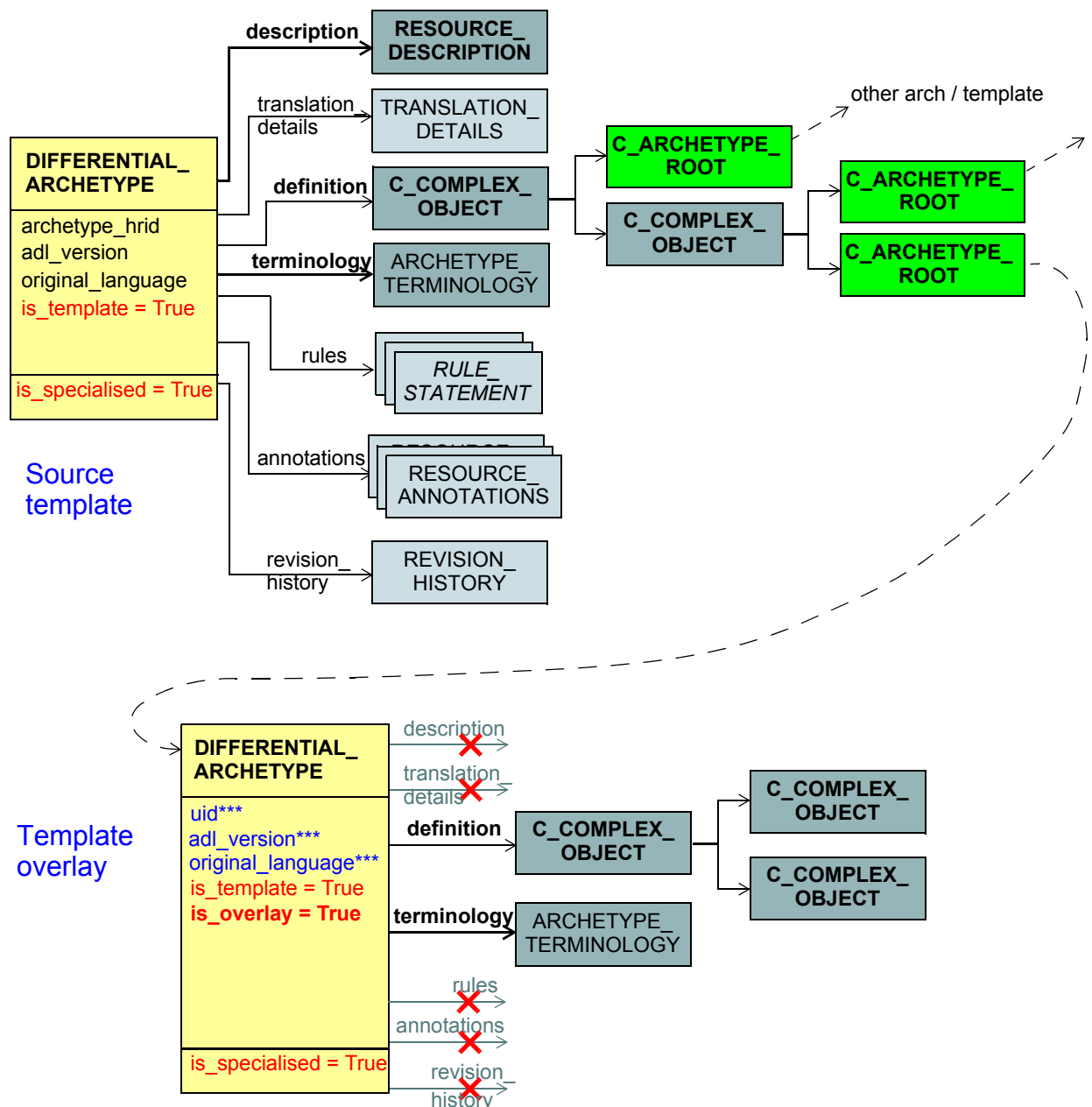


FIGURE 14 Source template instance structures

FIGURE 15 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.

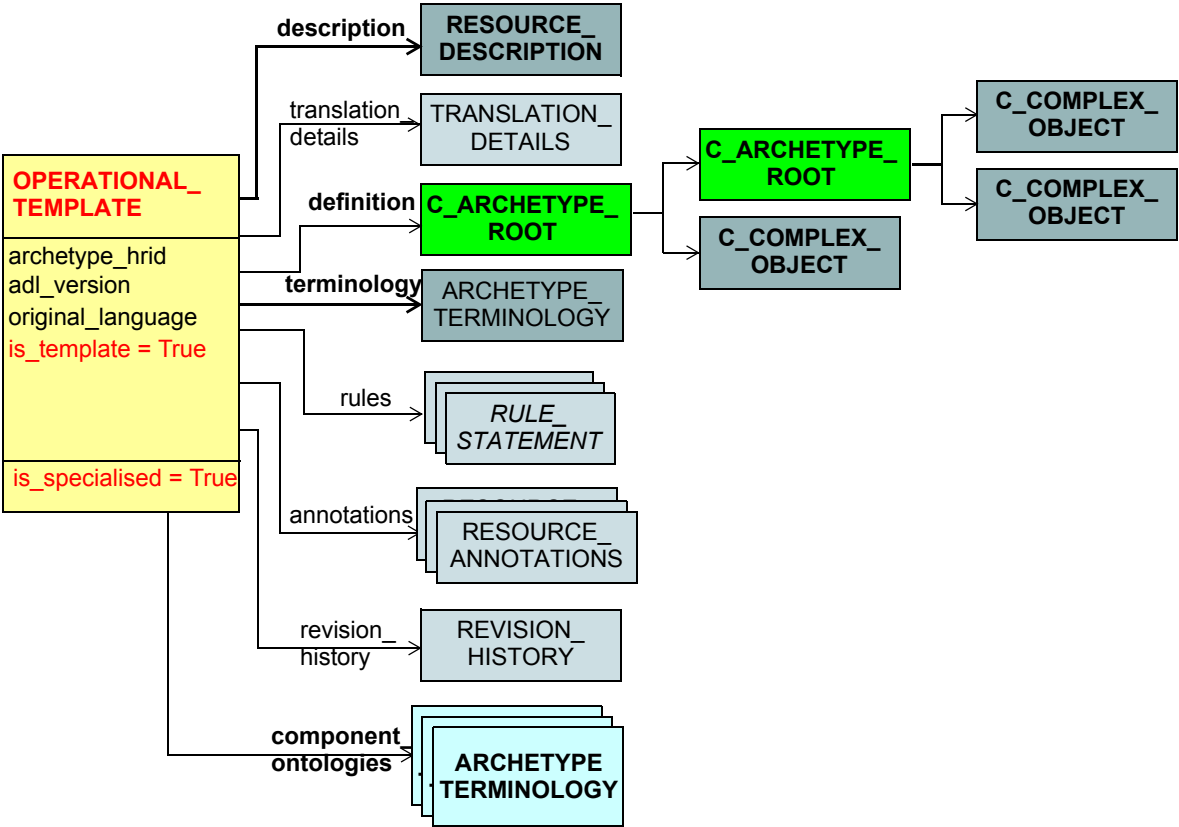


FIGURE 15 Operational template instance structure

5.2 Class Descriptions

5.2.1 ARCHETYPE Class

CLASS	ARCHETYPE (abstract)	
Purpose	Root object of an archetype. Defines semantics of identification, lifecycle, versioning, composition and specialisation.	
Inherit	AUTHORED_RESOURCE	
Attributes	Signature	Meaning
0..1	adl_version: String	ADL version if archteype was read in from an ADL shareable archetype.
0..1	urn: URN	Unique identifier of this logical archetype. Assigned at or after creation, and remains fixed for the life of the archetype.

CLASS	ARCHETYPE (abstract)	
0..1	instance_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.
0..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	archetype_id: ARCHETYPE_HRID	Identifier of this archetype.
0..1	parent_archetype_id: String	Archetype reference of the specialisation parent of this archetype, if applicable. May take the form of an archetype interface identifier, i.e. the identifier up to the major version only, or can be deeper.
1	is_template: Boolean	Indicates the type of artefact, i.e. archetype, or template. If not present (e.g. in serialised forms), assumed to be False.
1	is_overlay: Boolean	Indicates the artefact to be an overlay of an archetype or template; if True, <i>is_specialised</i> must also be True.
1	definition: C_COMPLEX_OBJECT	Root node of this archetype
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
1	concept_code: String <i>ensure</i> 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.

CLASS	ARCHETYPE (<i>abstract</i>)	
1	physical_paths: Set<String>	Set of language-independent paths extracted from archetype. Paths obey Xpath-like syntax and are formed from alternations of <code>C_OBJECT.node_id</code> and <code>C_ATTRIBUTE.rm_attribute_name</code> values.
1	logical_paths (a_lang: String): Set<String>	Set of language-dependent paths extracted from archetype. Paths obey the same syntax as <code>physical_paths</code> , but with <i>node_ids</i> replaced by their meanings from the terminology.
	is_specialised: Boolean <i>ensure</i> <i>Result</i> implies parent_archetype_hrid != Void	True if this archetype is a specialisation of another.
	specialisation_depth: Integer <i>ensure</i> <i>Result</i> = terminology. specialisation_depth	Specialisation depth of this archetype; larger than 0 if this archetype has a parent. Derived from the specialisation depth of <i>concept_code</i> .
Invariant	-- invariants for all archetypes Uid_validity: uid != Void implies not uid.is_empty Instance_uid_validity: instance_uid != Void implies not instance_uid.is_empty Template_validity: is_template implies is_specialised Overlay_validity: is_overlay implies (is_specialised and is_template) Original_language_valid: original_language != void and language != Void and then code_set (Code_set_id_languages).has_code (original_language) Concept_valid: terminology.has_term_code (concept_code) Definition_exists: definition != Void Terminology_exists: terminology != Void Specialisation_validity: is_specialised implies specialisation_depth > 0 Rules_valid: rules != Void implies not rules.is_empty -- template / overlay invariants Archetype_id_validity: is_overlay xor archetype_id != Void Template_uid_validity: is_overlay implies uid != Void Description_existence: is_overlay implies description = Void Annotations_existence: is_overlay implies annotations = Void Rules_existence: is_overlay implies rules = Void	

5.2.2 ARCHETYPE_HRID Class

CLASS	ARCHETYPE_HRID
Purpose	Human_readable identifier (HRID) for an archetype or template.

CLASS	ARCHETYPE_HRID	
Attributes	Signature	Meaning
0..1	namespace: <code>INTERNET_ID</code>	Reverse domain name namespace identifier.
1	rm_publisher: <code>String</code>	Name of the Reference Model publisher.
1	rm_closure: <code>String</code>	Name of the package in whose closure the <i>rm_class</i> class is found (there can be more than one possibility in a reference model).
1	rm_class: <code>String</code>	Name of the root class of this archetype.
1	concept_id: <code>String</code>	The short concept name of the archetype as used in the multi-axial <i>archetype_hrid</i> .
1	release_version: <code>String</code>	The full numeric version of this archetype consisting of 3 parts, e.g. 1.8.2. The <i>archetype_hrid</i> feature includes only the major version.
1	version_status: <code>VERSION_STATUS</code>	The status of the version.
Functions	Signature	Meaning
	interface_id: <code>String</code>	The ‘interface’ form of the HRID, i.e. down to the major version.
	physical_id: <code>String</code>	The ‘physical’ form of the HRID, i.e. with complete version information.
	version_id: <code>String</code>	Full version identifier string, based on <i>release_version</i> and <i>lifecycle</i> , e.g. 1.8.2-rc4.
	major_version: <code>String</code>	Major version of this archetype, extracted from <i>release_version</i> .
	minor_version: <code>String</code>	Minor version of this archetype, extracted from <i>release_version</i> .
	patch_version: <code>String</code>	Patch version of this archetype, extracted from <i>release_version</i> . Equivalent to patch version in patch version in semver.org sytem.
	uri: <code>String</code>	Generated URI based on <i>physical_id</i> .

CLASS	ARCHETYPE_HRID
Invariant	<p><i>Rm_publisher_validity</i>: rm_publisher != Void and then not rm_publisher.is_empty</p> <p><i>Rm_closure_validity</i>: rm_closure != Void and then not rm_closure.is_empty</p> <p><i>Rm_class_name_validity</i>: rm_class != Void and then not rm_class.is_empty</p> <p><i>Concept_id_validity</i>: concept_id != Void and then not concept_id.is_empty</p> <p><i>Base_version_validity</i>: release_version != Void and then not release_version.is_empty</p>

5.2.3 DIFFERENTIAL_ARCHETYPE Class

CLASS	DIFFERENTIAL_ARCHETYPE	
Purpose	Root object of the differential form of an archetype or template. Also called the 'source' form, as this is the form of an archetype created by an editor tool. For non-specialised archetypes, this is the almost same as the flat form (apart from the expansion of internal references in the flat form). For specialised archetypes, only the differences with respect to the parent are represented.	
Inherit	ARCHETYPE	
Attributes	Signature	Meaning
1 (redefined)	terminology : DIFFERENTIAL_ARCHETYPE_ TERMINOLOGY	Terminology, which takes the form of differential terminology, which contains only the terms defined or redefined in the current archetype.
Invariant		

5.2.4 FLAT_ARCHETYPE Class

CLASS	FLAT_ARCHETYPE	
Purpose	Root object of the inheritance-flattened form of an archetype or template.	
Inherit	ARCHETYPE	
Attributes	Signature	Meaning
1 (redefined)	terminology : FLAT_ARCHETYPE_ TERMINOLOGY	Terminology, which takes the form of flat terminology, which contains the terms defined or redefined in the current archetype as well as inherited terms, other than those overridden by the local definitions and redefinitions (which are replaced by the latter).
Invariant		

5.2.5 OPERATIONAL_TEMPLATE Class

CLASS	OPERATIONAL_TEMPLATE	
Purpose	Root object of an operational template. An operational template is derived from a template definition and the archetypes mentioned by that template by a process of flattening, and potentially removal of unneeded languages and terminologies.	
Use	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	FLAT_ARCHETYPE	
Attributes	Signature	Meaning
1 (redefined)	definition: C_ARCHETYPE_ROOT	Root node is replaced with a C_ARCHETYPE_ROOT. This has the effect that every interior node in the entire structure has a C_ARCHETYPE_ROOT above it.
1	component_terminologies: Hash <FLAT_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.
Functions	Signature	Meaning
	component_terminology (an_id: String): FLAT_ARCHETYPE_TERMINOLOGY	Terminology for archetype or template component with identifier <i>an_id</i> .
Invariant	<i>Is_specialised:</i> is_specialised <i>Is_template:</i> is_template <i>Definition_existence:</i> definition != Void <i>Component_terminologies_existence:</i> component_terminologies != Void	

5.3 Validity Rules

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

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 specialisation 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 accessibility; 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 description, 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.

6 Constraint Model Package

6.1 Overview

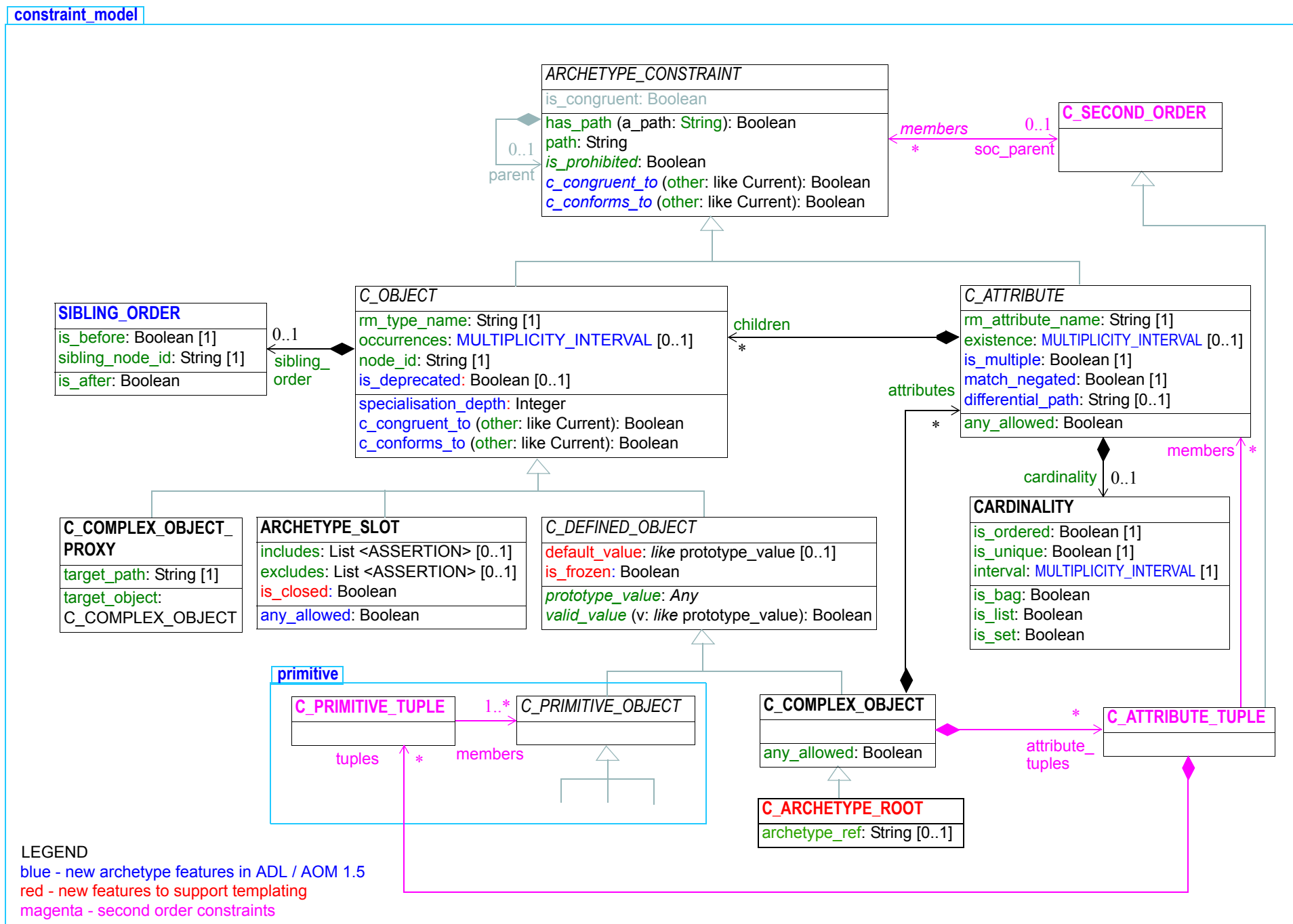
FIGURE 16 and FIGURE 17 illustrate the object model of constraints used in an archetype definition. This model is completely generic, and is designed to express the semantics of constraints on instances of classes which are themselves described in any orthodox object-oriented formalism, such as UML. 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.



primitive

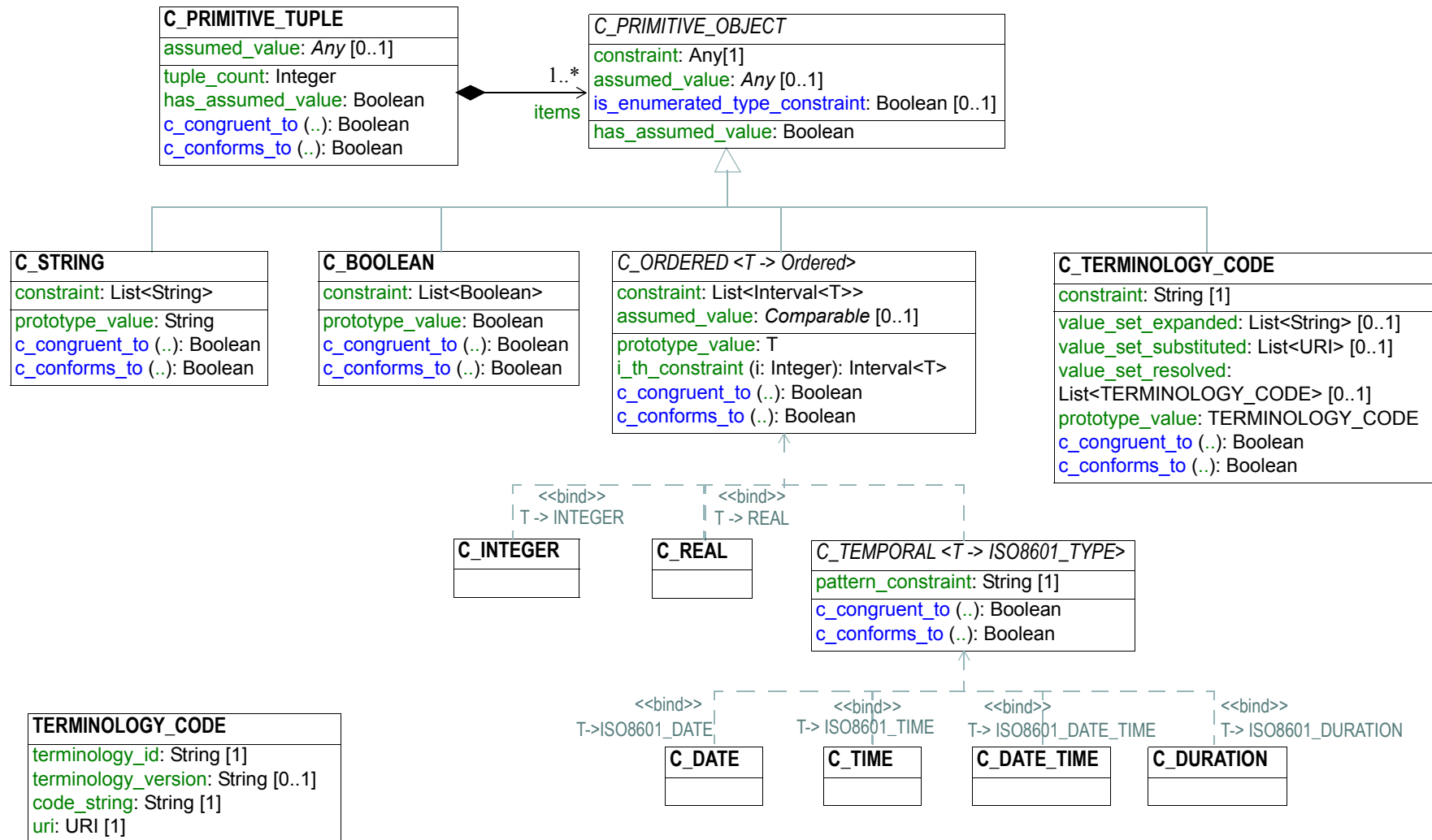


FIGURE 17 The openehr.am.archetype.primitive Package

6.2 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 [9], 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.

6.2.1 All Node Types

Path Functions

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

Conformance Functions

All node types include two functions that formalise the notion of *conformance* of a specialised archetype to a parent archetype. Both functions take an argument which 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 Class Definitions on page 50.

6.2.2 Attribute Node Types

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

- *is_multiple*: a flag that indicates whether the *C_ATTRIBUTE* is constraining a multiply-valued (i.e. container) RM attribute or a single-valued one;
- *existence*: whether the corresponding instance (defined by the *rm_attribute_name* attribute) 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 18 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

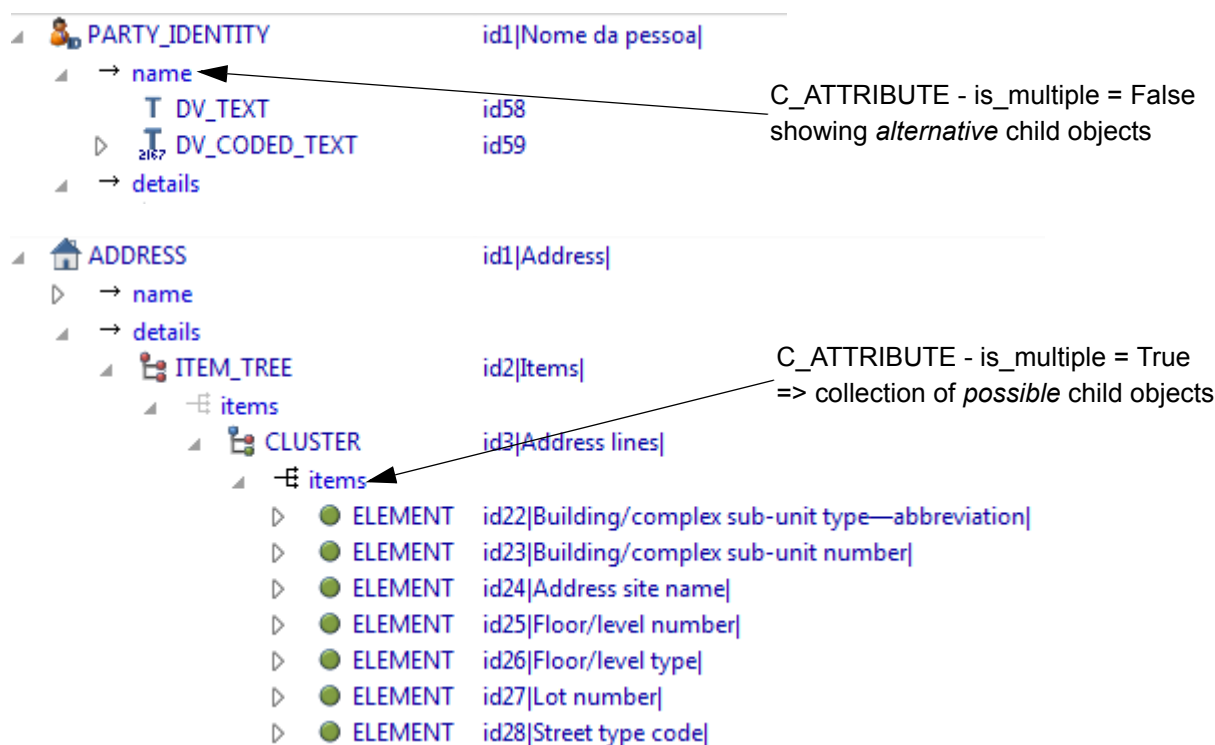


FIGURE 18 Single and Multiple-valued C_ATTRIBUTES

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.

6.2.3 Object Node Types

Node_id and Paths

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

- it allows archetype object constraint nodes to be individually identified, and in particular, guarantees sibling node unique identification;
- it provides a code to which a human-understanding terminology definition can be attached, as well as potentially a terminology binding.

The existence of *node_ids* in an archetype allows archetype paths to be created, which refer to each node. Every node in the archetype needs a *node_id*, but only *node_ids* for nodes under container attributes 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.

6.2.3.1 Defined Object Nodes (C_DEFINED_OBJECT)

The `C_DEFINED_OBJECT` subtype corresponds to the category of `C_OBJECT`s 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 archetype-based software to build a 'prototype' object from an archetype which can serve as the initial version of the object being constrained, assuming it is being created new by user activity (e.g. via a GUI application). Implementation of this function will usually involve use of reflection libraries or similar.

Default_value

This attribute allows a user-specified default value to be defined within an archetype. The *default_value* object 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.

6.2.3.2 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_OBJECT*s.

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.

6.2.3.3 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

Primitive type	Primitive constrainer type	Explanation
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, representing 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.

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.

Enumeration types in the reference model are assumed to have semantics expected in UML, and mainstream programming languages, i.e. to be a distinct type based on a primitive type, normally Integer or String. Each such type consists of a set of values from the domain of its underlying type, thus, a set of Integer, String or other primitive values. Each of these values is assumed to be named in the manner of a symbolic constant. Although strictly speaking UML 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.

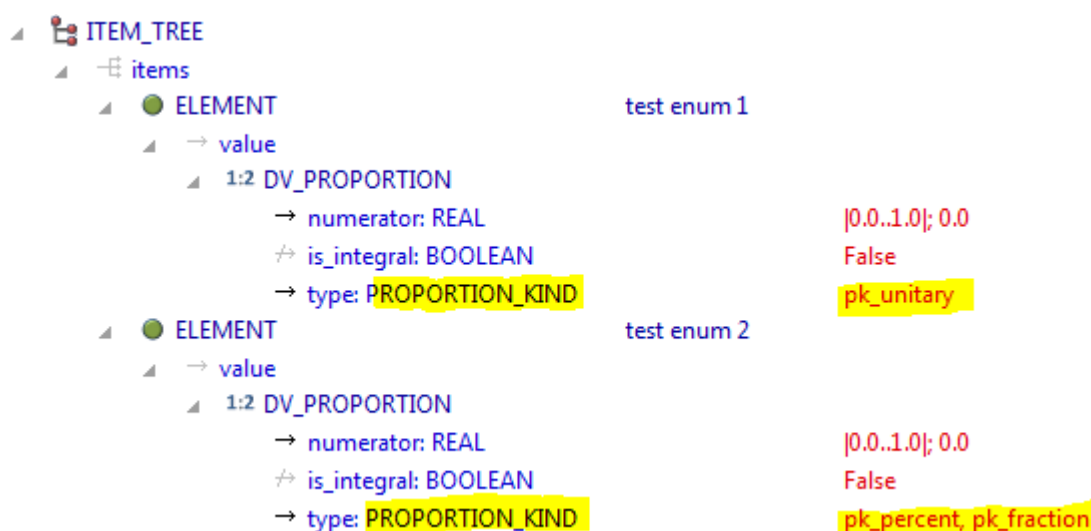


FIGURE 19 Enumerated type constraint

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.

6.3 Second Order Constraints

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

Some constraints however do not fit directly within the object/attribute/object hierarchy scheme, and are considered ‘second order constraints’ in the archetype formalism. The ‘rule’ constraints (‘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 section 7 on page 77.

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.

6.3.1 Tuple Constraints

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

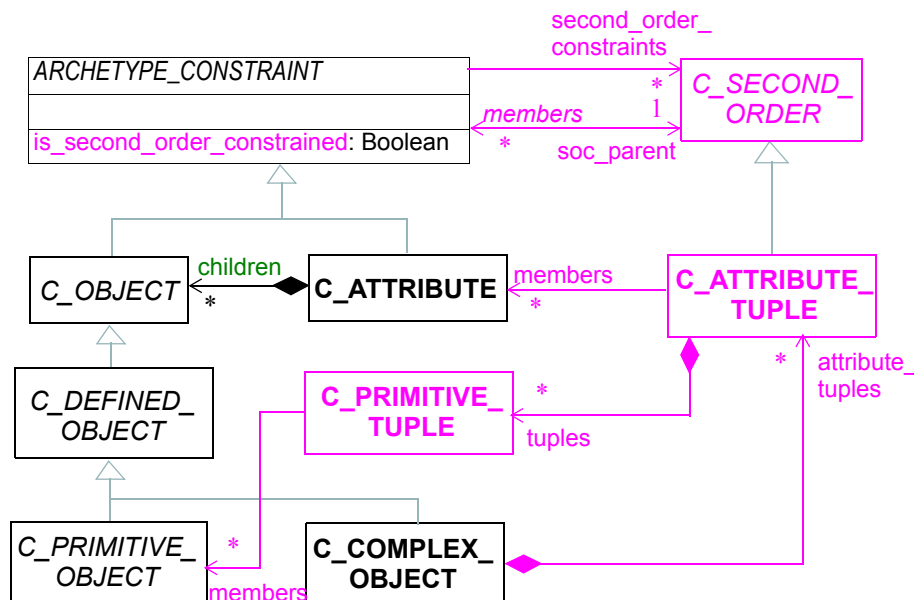


FIGURE 20 Tuple Constraint AOM Additions

In this model, the type `C_ATTRIBUTE_TUPLE` groups the co-constrained `C_ATTRIBUTES` 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.

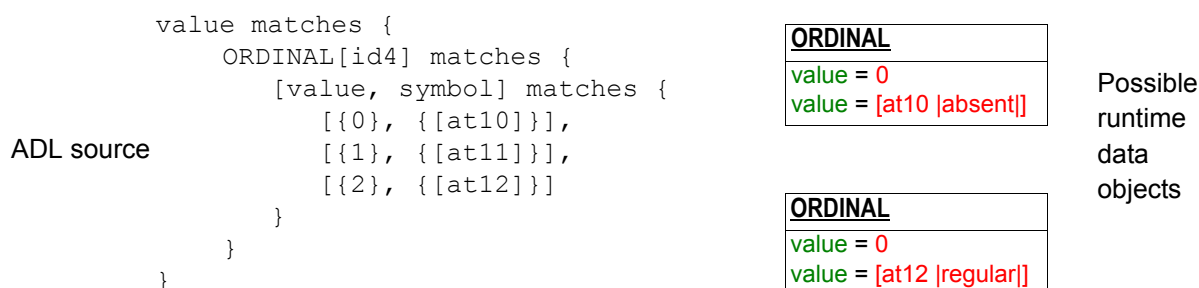
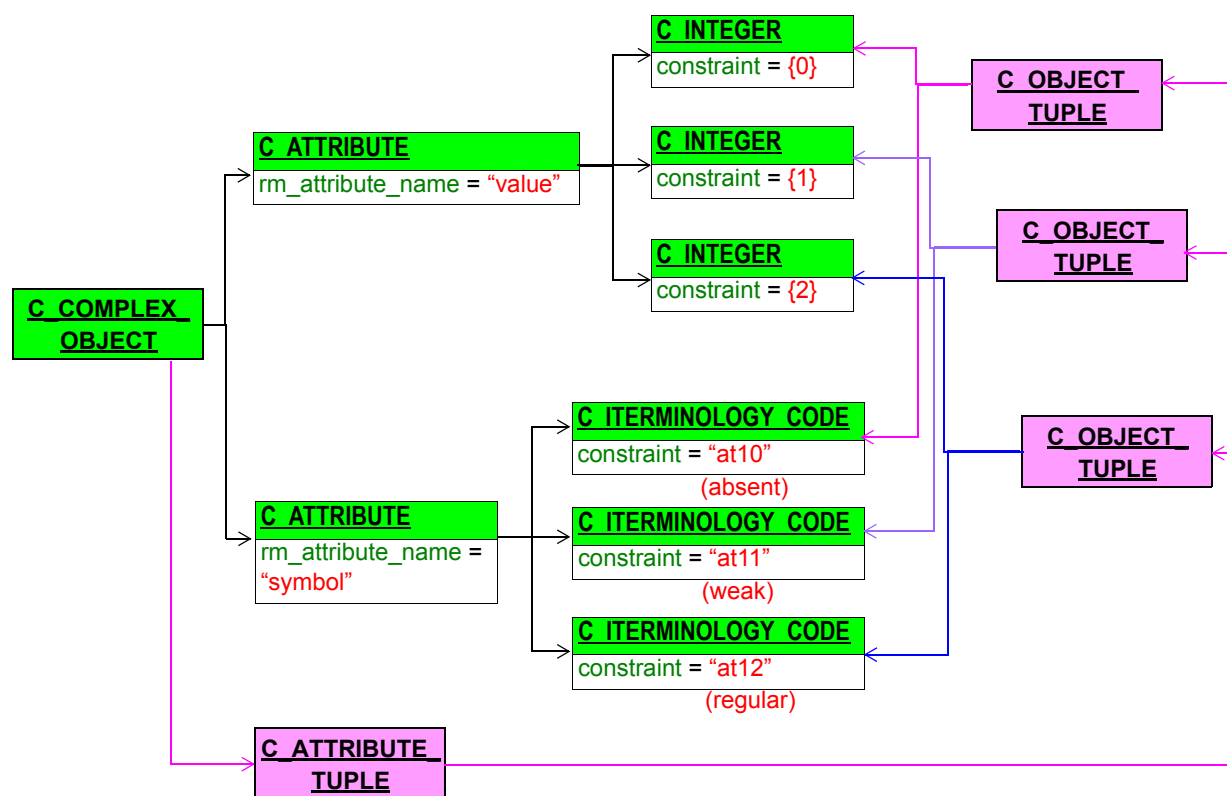


FIGURE 21 Example of tuple constraint

6.3.2 Group Constraint

Within a container attribute, it is not uncommon to need to define and constrain sub-groups within the overall list of objects in the container. Two types of constraint are needed:

- to control the total number of elements allowed in the group at runtime, from the total defined;
- to control the number of times the group repeats.

The first constraint is achieved by defining a cardinality of the group. A cardinality of '*' means that in the data, the group may contain any number of objects, each conforming to one of the object types

defined within that group in the archetype. A cardinality of '1' specifies a 'choice' of 1 of N items in the group.

Control of repetition of the group as a whole is achieved with an occurrences constraint on the group, indicating the number of times the group can repeat within the data.

FIGURE 22 illustrates the group constraint part of the `constraint_model` package. A grouping constraint is represented as a `C_OBJECT_GROUP` instance attached to a `C_ATTRIBUTE`. The primary representation of the group is as a list of `C_OBJECT` members of the `C_ATTRIBUTE.children` list. The children actually referenced are computed by the functions *lower_child* and *upper_child*.

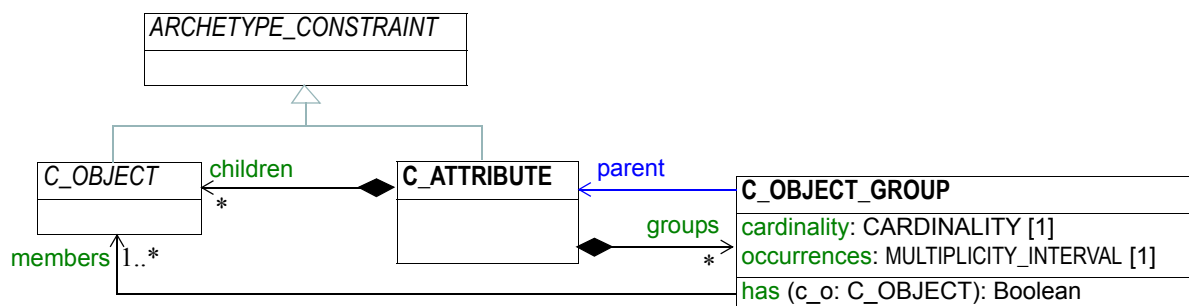


FIGURE 22 Group constraint

The integer range representation is used because it allows the validity conditions on `C_ATTRIBUTE` for groups to be easily stated. If there are multiple groups for a given container, all have to obey invariants that state either mutual exclusion or proper containment, i.e. no overlapping.

6.3.3 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.

6.4 C_OBJECT Substitutions

The `C_OBJECT` types defined in FIGURE 16 are reproduced below, with concrete types that may actually occur in archetypes shown in dark yellow / non-italic.

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.

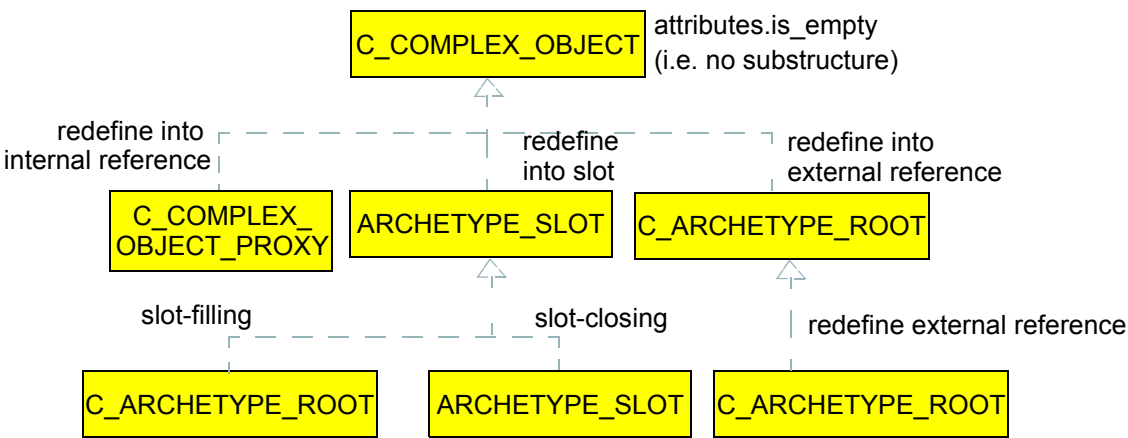


FIGURE 23 C_OBJECT legal redefinitions

- 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:
 - An `ARCHETYPE_SLOT` node, indicating that the object will be defined by other archetypes;
 - A `C_ARCHETYPE_ROOT`, indicating that the node will be defined by another specific archetype;
 - A `C_COMPLEX_OBJECT_PROXY` node.
- 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.

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

6.5 Class Definitions

6.5.1 ARCHETYPE_CONSTRAINT Class

CLASS	ARCHETYPE_CONSTRAINT (abstract)	
Purpose	Archetype equivalent to LOCATABLE class in openEHR Common reference model. Defines common constraints for any inheritor of LOCATABLE in any reference model.	
Abstract	Signature	Meaning
	<i>is_prohibited</i> : Boolean	True if this node is prohibited.

CLASS	ARCHETYPE_CONSTRAINT (abstract)	
	<i>c_conforms_to</i> (other: like Current): Boolean require other != Void	True if constraints represented by this node, ignoring any sub-parts, are narrower or the same as <i>other</i> . Typically used during validation of specialised archetype nodes.
	<i>c_congruent_to</i> (other: like Current): Boolean require other != Void	True if constraints represented by this node contain no further redefinitions with respect to the node <i>other</i> , with the exception of <i>node_id</i> redefinition in C_OBJECT nodes. Typically used to test if an inherited node locally contains any constraints.
Attributes	Signature	Meaning
0..1 (non-persistent)	parent: ARCHETYPE_CONSTRAINT	Parent node in hierarchy. Void if root node.
0..1 (non-persistent)	soc_parent: C_SECOND_ORDER	Where applicable, parent second-order constraint node for which this object is in the <i>members</i> .
0..1	second_order_constraints: Set <C_SECOND_ORDER>	Generate set of all second order constraints applying to this object and immediate children.
Functions	Signature	Meaning
	is_root: Boolean	True if this node is the root of constraint structure.
	path: String	Path of this node relative to root of archetype.
	has_path (a_path: String): Boolean require a_path != Void	True if the relative path <i>a_path</i> exists at this node.
	is_second_order_constrained: Boolean	True if <i>soc_parent</i> != Void or <i>parent.soc_parent</i> != Void
Invariant	path_exists: path != Void	

6.5.2 C_ATTRIBUTE Class

CLASS	C_ATTRIBUTE	
Purpose	The model of a constraint on a reference model attribute (including computed attributes).	
Inherit	ARCHETYPE_CONSTRAINT	
Attributes	Signature	Meaning
1	rm_attribute_name: String	Reference model attribute within the enclosing type represented by a C_OBJECT.
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 redefined parts of a specialised archetype to be expressed, at the path where they occur.
0..1	existence: MULTIPLICITY_INTERVAL	Constraint 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.
1	match_negated: Boolean	True if the match operator on this attribute is negated, i.e. the constraint structure below this C_ATTRIBUTE is <i>not</i> to be matched by the data rather than to be matched.
0..1	cardinality: CARDINALITY	Cardinality of this attribute constraint, if it constrains a container attribute.
0..1	groups: List<C_OBJECT_GROUP>	Group constraints applying to the children of this attribute.
Functions	Signature	Meaning
	is_multiple: Boolean	True if this node logically represents a container attribute.

CLASS	C_ATTRIBUTE	
	is_single : Boolean	True if this node logically represents a single-valued attribute.
	is_mandatory : Boolean	True if this node mandates the existence of its corresponding RM attribute in data.
	is_prohibited : Boolean	True if this node prohibits the existence of its corresponding RM attribute in data. Used to remove nodes in templates.
	rm_attribute_path : String	Path of this attribute with respect to owning C_OBJECT, including differential path where applicable.
(redefined)	path : String	If <i>has_differential_path</i> , returns <i>rm_attribute_path</i> , else returns <i>path</i> as defined in ARCHETYPE_CONSTRAINT.
(effected)	c_conforms_to (other: like Current): Boolean	True if <i>existence</i> of this object conforms to <i>existence</i> of other, and if the nodes are both container nodes, if <i>cardinality</i> conforms.
(effected)	c_congruent_to (other: like Current): Boolean	True if <i>existence</i> is Void, and if this node and <i>other</i> are both container attribute nodes, if <i>cardinality</i> is Void.
	existence_conforms_to (other: like Current): Boolean <i>require</i> other /= Void	True if the existence of this node conforms to existence of node <i>other</i> ; returns True if the existence of this attribute is Void.
	cardinality_conforms_to (other: like Current): Boolean <i>require</i> other /= Void	True if the cardinality of this node conforms to cardinality of node <i>other</i> , returns True if the cardinality of this attribute is Void.
	any_allowed : Boolean <i>ensure</i> children.is_empty implies Result	True if no child nodes, i.e. children.count = 0
	has_differential_path : Boolean	True if differential_path is not Void..
	occurrences_lower_sum : INTEGER	Sum of lower bounds of occurrences of all child objects.
	minimum_child_count : INTEGER	Notional minimum number of possible children, counting 1 for each mandatory child and 1 further child to cover all optional children.

CLASS	C_ATTRIBUTE	
	groups_valid : Boolean	True if <i>groups</i> is valid.
Invariant	<p><i>Rm_attribute_name_valid</i>: <i>rm_attribute_name</i> /= Void and then not <i>rm_attribute_name.is_empty</i></p> <p><i>Existence_valid</i>: <i>existence</i> /= Void implies (<i>existence.lower</i> >= 0 and <i>existence.upper</i> <= 1)</p> <p><i>Children_validity</i>: <i>any_allowed</i> xor <i>children</i> /= Void</p> <p><i>Cardinality_valid</i>: <i>cardinality</i> /= Void implies <i>is_multiple</i></p> <p><i>Children_occurrences_lower_sum_validity</i>: (<i>cardinality</i> /= Void and then not <i>cardinality.interval.upper_unbounded</i>) implies <i>occurrences_lower_sum</i> <= <i>cardinality.interval.upper</i></p> <p><i>Children_orphans_validity</i>: (<i>cardinality</i> /= Void and then not <i>cardinality.interval.upper_unbounded</i>) implies <i>minimum_child_count</i> <= <i>cardinality.interval.upper</i></p> <p><i>Differential_path_valid</i>: <i>differential_path</i> /= Void implies not <i>differential_path.is_empty</i></p> <p><i>Alternatives_valid</i>: <i>not is_multiple</i> implies <i>children.for_all</i> (<i>co</i>: C_OBJECT <i>co.occurrences.upper</i> <= 1)</p> <p><i>Groups_valid</i>: <i>groups</i> /= Void implies <i>is_multiple</i></p> <p><i>Child_occurrences_validity</i>: <i>cardinality</i> /= Void implies <i>cardinality.interval.intersects</i> (<i>occurrences_total_range</i>)</p>	

6.5.2.1 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
       (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

    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
end

```

6.5.2.2 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 \leq 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.

6.5.2.3 Groups validity algorithm

The following pseudo-code expresses the validity of the *groups* attribute, where it is non-Void:

```
groups.for_all ( TBC )
To Be Determined:
```

6.5.3 CARDINALITY Class

CLASS	CARDINALITY	
Purpose	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	is_ordered: Boolean	True if the members of the container attribute on which this cardinality is defined are ordered.
1	is_unique: Boolean	True if the members of the container attribute on which this cardinality is defined are unique.

CLASS	CARDINALITY	
1	interval: MULTIPLICITY_INTERVAL	The interval of this cardinality.
Functions	Signature	Meaning
	is_set: Boolean <i>ensure</i> <i>Result</i> = not is_ordered and is_unique	True if the semantics of this cardinality represent a set, i.e. unordered, unique membership.
	is_list: Boolean <i>ensure</i> <i>Result</i> = is_ordered and not is_unique	True if the semantics of this cardinality represent a list, i.e. ordered, non-unique membership.
	is_bag: Boolean <i>ensure</i> <i>Result</i> = not is_ordered and not is_unique	True if the semantics of this cardinality represent a bag, i.e. unordered, non-unique membership.
Invariant	<i>Validity:</i> not interval.lower_unbounded	

6.5.4 C_OBJECT Class

CLASS	C_OBJECT (abstract)	
Purpose	Abstract model of constraint on any kind of object node.	
Inherit	ARCHETYPE_CONSTRAINT	
Attributes	Signature	Meaning
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.

CLASS	C_OBJECT (abstract)	
1	node_id: String	Semantic identifier of this node, used to distinguish sibling nodes. All nodes must have a <i>node_id</i> ; for nodes under a container C_ATTRIBUTE, the id must be an id-code must be defined in the archetype terminology. For valid structures, all node ids are id-codes. For structures in intermediate state, or structures originating from legacy archetypes, the special Anonymous_node_id value can be used. For C_PRIMITIVE_OBJECTs, it will have the special value Primitive_node_id.
0..1	parent: C_ATTRIBUTE	C_ATTRIBUTE that owns this C_OBJECT.
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 <i>is_multiple</i> = True.
0..1	is_deprecated: Boolean	True if this node and by implication all sub-nodes are deprecated for use.
Functions	Signature	Meaning
	is_addressable: Boolean	True if this node has a node_id that is not the Anonymous_node_id value.
	is_prohibited: Boolean <i>ensure</i> Result = occurrences /= Void and then occurrences.is_prohibited	True if this node is prohibited due to its occurrences being set to 0..0.
(effected)	c_conforms_to (other: <i>like</i> Current): Boolean	Returns True only when: - <i>rm_type_name</i> is the same or a subtype of <i>rm_type_name</i> of other; - occurrences is same (= Void) or a sub-interval - node_id is the same, or redefined to a legal code at the level of the owning archetype

CLASS	C_OBJECT (abstract)	
(effected)	c_congruent_to (other: <i>like Current</i>): Boolean	Returns True if: - <i>rm_type_name</i> is identical - <i>occurrences</i> is Void - <i>sibling_order</i> is Void - <i>node_id</i> is conformant (i.e. same of specialisation of other's <i>node_id</i>) - this object is the sole redefinition of other
	rm_type_conforms_to (other: <i>like Current</i>): Boolean <i>require</i> other != Void	True if this node <i>rm_type_name</i> conforms to other. <i>rm_type_name</i> by either being equal, or by being a subtype, according to the underlying reference model.
	occurrences_conforms_to (other: <i>like Current</i>): Boolean <i>require</i> other != Void	True if this node occurrences conforms to other.occurrences. returns True if occurrences of this object is Void.
	node_id_conforms_to (other: <i>like Current</i>): Boolean <i>require</i> other != Void	True if this node id conforms to other. <i>node_id</i> .
	specialisation_depth : Integer	Level of specialisation of this archetype node, based on its <i>node_id</i> . 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 <i>node_id</i> code. If <i>node_id</i> 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 <i>node_id</i> .
Invariant	<i>Rm_type_name_valid</i> : <i>rm_type_name</i> != Void and then not <i>rm_type_name.is_empty</i> <i>Node_id_valid</i> : <i>node_id</i> != Void and then not <i>node_id.is_empty</i> <i>Occurrences_validity</i> : (<i>occurrences</i> != Void and <i>parent</i> != Void and <i>parent.is_single</i>) implies <i>occurrences.upper</i> <= 1 <i>Sibling_order_validity</i> : <i>sibling_order</i> != Void implies <i>specialisation_depth</i> > 0 and <i>parent.is_multiple</i>	

6.5.4.1 Occurrences inferencing rules

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

inferred from the reference model according to the following algorithm, where `c_object` represents any object node in an archetype.

```

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

6.5.4.2 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_equal (other.rm_type_name) and
              occurrences = Void and
              node_id_conforms_to (other) and sibling_order = Void
  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 := node_id.starts_with (other.node_id)
  end
end

```

6.5.4.3 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.

VC OCD: 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}.

VC OID: object node identifier validity: every object node must have a node identifier.

VC OSU: object node identifier validity: every object node must be unique within the archetype.

The following validity rules govern C_OBJECTs in specialised archetypes.

VS ONT: 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: C_COMPLEX_OBJECT with no child attributes may be redefined by C_COMPLEX_OBJECT_PROXY, C_ARCHETYPE_ROOT and ARCHETYPE_SLOT; ARCHETYPE_SLOT may be redefined by C_ARCHETYPE_ROOT (i.e. 'slot-filling'). See also validity rules VSUNT and VSCNR.

VS ONCT: 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: **VS ONIR: 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 specialised 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, specialised 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.

VSONPO: specialised archetype object node prohibited occurrences validity: the occurrences of a non-redefined 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.

6.5.5 SIBLING_ORDER Class

CLASS	SIBLING_ORDER	
Purpose	Defines the order indicator that can be used on an 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	is_before: Boolean	True if the order relationship is 'before', if False, it is 'after'.
1	sibling_node_id: String	Node identifier of sibling before or after which this node should come.
Functions	Signature	Meaning

CLASS	SIBLING_ORDER	
	is_after : Boolean	True if the order relationship is 'after', computed as the negation of <i>is_before</i> .
Invariant	<i>sibling_node_id_validity</i> : sibling_node_id /= Void	

6.5.6 C_DEFINED_OBJECT Class

CLASS	C_DEFINED_OBJECT (abstract)	
Purpose	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	
Abstract	Signature	Meaning
	<i>prototype_value</i> : Any	Generate a prototype value from this constraint object
	<i>valid_value</i> (a_value: like prototype_value): Boolean <i>require</i> a_value /= Void	True if a_value is valid with respect to constraint expressed in concrete instance of this type.
Attributes	Signature	Meaning
0..1	default_value : <i>like</i> prototype_value	Default value set in a template, and present in an operational template. Generally limited to leaf and near-leaf nodes.
0..1	is_frozen : Boolean	True if this node is closed for further redefinition. Any child nodes defined as siblings are considered to exhaustively represent the possible value space of this original parent node.
Functions	Signature	Meaning
	has_default_value : Boolean	True if there is a default value
Invariant	<i>Default_value_valid</i> : has_default_value implies default_value.conforms_to_type (rm_type_name) and valid_value (default_value)	

6.5.7 C_COMPLEX_OBJECT Class

CLASS	C_COMPLEX_OBJECT	
Purpose	Constraint on complex objects, i.e. any object that consists of other object constraints.	

CLASS	C_COMPLEX_OBJECT	
Inherit	C_DEFINED_OBJECT	
Attributes	Signature	Meaning
0..1	attributes: Set<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 tuple constraints.
Functions	Signature	Meaning
	any_allowed: Boolean <i>ensure</i> Result = attributes.is_empty	True if any value of the reference model type being constrained is allowed.
Invariant	<i>Prohibited_validity:</i> not (any_allowed and is_prohibited) <i>Any_allowed_validity:</i> any_allowed xor not attributes.is_empty <i>Attributes_valid:</i> attributes != Void <i>Tuples_valid:</i> attribute_tuples != Void implies attribute_tuples.for_all (cat: C_ATTRIBUTE_TUPLE cat.members.for_all (ca: C_ATTRIBUTE) attributes.has(ca)))	

6.5.7.1 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.

6.5.8 C_ARCHETYPE_ROOT Class

CLASS	C_ARCHETYPE_ROOT
Purpose	A specialisation of C_COMPLEX_OBJECT whose <i>node_id</i> attribute is an archetype identifier rather than the normal internal node code (i.e. at-code).

CLASS	C_ARCHETYPE_ROOT	
Use	<p>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.</p> <p>For a new external reference, the <i>node_id</i> is set in the normal way, i.e. with a new code for the specialisation level of the archetype.</p> <p>For a slot-filler or a redefined external reference, the <i>node_id</i> is set to a specialised version of the <i>node_id</i> of the node being specialised, allowing matching to occur during flattening.</p> <p>In all uses within <i>source</i> archetypes and templates, the <i>children</i> attribute is Void.</p> <p>In an operational template, the <i>node_id</i> is converted to the <i>archetype_ref</i>, and the structure contains the result of flattening any template overlay structure and the underlying flat archetype.</p>	
Inherit	C_COMPLEX_OBJECT	
Attributes	Signature	Meaning
0..1	archetype_ref : String	Reference to archetype is being used to fill a slot or redefine an external reference. Typically an ‘interface’ archetype id, i.e. identifier with partial version information.
Invariant	<i>Archetype_ref_validity</i> : archetype_ref != Void	

6.5.8.1 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.

6.5.9 ARCHETYPE_SLOT Class

CLASS	ARCHETYPE_SLOT	
Purpose	Constraint describing a 'slot' point at which one or more other archetypes matching the slot constraint can be included.	
Inherit	C_OBJECT	
Attributes	Signature	Meaning
0..1	includes : Set<ASSERTION>	List of constraints defining other archetypes that could be included at this point.
0..1	excludes : Set<ASSERTION>	List of constraints defining other archetypes that cannot be included at this point.
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 <i>ensure</i> Result = not (has_includes or has_excludes) and not is_closed	True if no constraints stated, and slot is not closed.
Invariant	<i>includes_valid</i> : includes != Void implies not includes.is_empty <i>excludes_valid</i> : excludes != Void implies not excludes.is_empty	

6.5.9.1 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_SLOT`s defined as the specialisation of a slot in a parent archetype:

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.

VDSSR: slot redefinition child validity: an `ARCHETYPE_SLOT` node in a specialised archetype may redefine a `C_COMPLEX_OBJECT` if the latter contains no sub-structure (i.e. `attributes.is_empty`).

6.5.10 C_COMPLEX_OBJECT_PROXY Class

CLASS	C_COMPLEX_OBJECT_PROXY	
Purpose	<p>A constraint defined by proxy, using a reference to an object constraint defined elsewhere in the same archetype.</p> <p>Note that since this object refers to another node, there are two objects with available occurrences values. The local <i>occurrences</i> value on an <code>C_COMPLEX_OBJECT_PROXY</code> should always be used if set. When setting this from a serialised form, if no occurrences is mentioned, the target occurrences should be used; otherwise the locally specified occurrences should be used as normal.</p>	
Inherit	<code>C_OBJECT</code>	
Attributes	Signature	Meaning
1	target_path: String	Path to target object node under same ultimate root object.
Functions	Signature	Meaning
	target_object: <code>C_COMPLEX_OBJECT</code>	Reference to object found at path 'target_path'.
Invariant	<p>Consistency: not any_allowed</p> <p>Target_path_valid: target_path != Void and then not target_path.is_empty</p> <p>-- and then ultimate_root.has_path(target_path)</p>	

6.5.10.1 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.

VSUNC: use_node specialisation child validity: a C_COMPLEX_OBJECT_PROXY node in a specialised archetype may redefine a C_COMPLEX_OBJECT if the latter contains no sub-structure (i.e. attributes.is_empty).

6.5.11 C_PRIMITIVE_OBJECT Class

CLASS	C_PRIMITIVE_OBJECT (abstract)	
Purpose	Abstract parent of classes defining constraints on primitive types.	
Inherit	C_DEFINED_OBJECT	
Attributes	Signature	Meaning
1	constraint: Any	Primary representation of the constraint. Actual type is set in descendants.
0..1	assumed_value: <i>like</i> prototype_value	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.
Functions	Signature	Meaning
	has_assumed_value: Boolean	True if there is an assumed value
Invariant	Representation_validity: constraint != Void Assumed_value_valid: has_assumed_value implies assumed_value.conforms_to_type (rm_type_name) and valid_value (assumed_value)	

6.5.11.1 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.

6.5.12 C_BOOLEAN Class

CLASS	C_BOOLEAN	
Purpose	Constraint on instances of Boolean.	
Use	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
(redefined)	constraint: List<Boolean>	Boolean constraint - a list of Boolean values.
Functions	Signature	Meaning
	true_valid: Boolean	True if the value True is allowed
	false_valid: Boolean	True if the value False is allowed
(redefined)	prototype_value: Boolean	A generated prototype value from this constraint object.
Invariant	<i>Binary_consistency:</i> true_valid <i>or</i> false_valid <i>Prototype_value_consistency:</i> .value <i>and</i> true_valid <i>or else not</i> .value <i>and</i> false_valid	

6.5.13 C_STRING Class

CLASS	C_STRING	
Purpose	Constraint on instances of STRING.	
Inherit	C_PRIMITIVE_OBJECT	
Attributes	Signature	Meaning
(redefined)	constraint: List<String>	String constraint - a list of literal strings and / or regular expression strings delimited by the '/' character.
Functions	Signature	Meaning

CLASS	C_STRING	
(redefined)	prototype_value: String	A generated prototype value from this constraint object.
	is_pattern: Boolean	True if <i>pattern</i> is not Void.
Invariant		

6.5.14 C_ORDERED Class

CLASS	C_ORDERED <T -> ORDERED> (abstract)	
Purpose	<p>Abstract parent of primitive constrainer classes based on <code>ORDERED</code> base types, i.e. types like <code>Integer</code>, <code>Real</code>, and the <code>Date/Time</code> 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.</p> <p>In its simplest form, the <i>constraint</i> accessor returns just a single point Interval <T> object, representing a single value.</p> <p>The next simplest form is a single proper Interval <T> (i.e. normal two-sided or half-open interval).</p> <p>The most complex form is a list of any combination of point and proper intervals.</p>	
Inherit	C_PRIMITIVE_OBJECT	
Attributes	Signature	Meaning
(redefined)	constraint: List<Interval<T>>	Constraint on any Ordered type - a list of Intervals, including 'point' intervals.
Functions	Signature	Meaning
(redefined)	prototype_value: T	A generated prototype value from this constraint object.
Invariant		

6.5.15 C_INTEGER Class

CLASS	C_INTEGER	
Purpose	Constraint on instances of <code>Integer</code> .	
Inherit	C_ORDERED <Integer>	
Invariant		

6.5.16 C_REAL Class

CLASS	C_REAL
Purpose	Constraint on instances of <i>Real</i> .
Inherit	C_ORDERED <Real>
Invariant	

6.5.17 C_TEMPORAL Class

CLASS	C_TEMPORAL <T -> ISO8601_TYPE> (abstract)	
Purpose	Abstract parent of C_ORDERED types whose base type is an ISO date/time type.	
Inherit	C_ORDERED <T -> ISO8601_TYPE>	
Attributes	Signature	Meaning
0..1	pattern: String	Optional alternative constraint in the form of a pattern.
Invariant		

6.5.18 C_DATE Class

CLASS	C_DATE	
Purpose	<p>ISO 8601-compatible constraint on instances of <i>Date</i> in the form either of a set of validity values, or else date ranges based on the C_ORDERED <i>list</i> constraint.</p> <p>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).</p>	
Use	Date ranges are probably only useful for historical dates.	
Inherit	C_TEMPORAL <ISO8601_DATE>	
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.

CLASS	C_DATE
Invariant	<i>Pattern_validity</i> : pattern /= Void implies valid_iso8601_date_constraint_pattern(pattern)

6.5.19 C_TIME Class

CLASS	C_TIME	
Purpose	<p>ISO 8601-compatible constraint on instances of <code>Time</code> in the form either of a set of validity values, or else date ranges based on the <code>C_ORDERED list</code> constraint.</p> <p>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).</p>	
Inherit	C_TEMPORAL <ISO8601_TIME>	
Functions	Signature	Meaning
(redefined)	prototype_value : ISO8601_TIME	A generated prototype value from this constraint object.
	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	<i>Pattern_validity</i> : pattern /= Void implies valid_iso8601_time_constraint_pattern(pattern)	

6.5.20 C_DATE_TIME Class

CLASS	C_DATE_TIME	
Purpose	<p>ISO 8601-compatible constraint on instances of <code>Date_Time</code>. 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-MM-DDTHH:MM:xx” (date/time, seconds not allowed).</p>	
Inherit	C_TEMPORAL <ISO8601_DATE_TIME>	
Functions	Signature	Meaning

CLASS	C_DATE_TIME	
	month_validity: VALIDITY_KIND	Validity of month in constrained date.
	day_validity: VALIDITY_KIND	Validity of day in constrained date.
	hour_validity: VALIDITY_KIND	Validity of hour in constrained time.
	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_date_time_constraint_pattern(pattern)	

6.5.21 C_DURATION Class

CLASS	C_DURATION	
Purpose	ISO 8601-compatible constraint on instances of <i>Duration</i> . In ISO 8601 terms, constraints might are of the form “PWD” (weeks and/or days), “PDTHMS” (days, hours, minutes, seconds) and so on. In official ISO 8601:2004, the ‘W’ (week) designator cannot be mixed in; allowing it is an <i>openEHR</i> -wide exception. The allowed patterns are: P[Y y][M m][D d][T[H h][M m][S s]] and P[W w].	
Inherit	C_TEMPORAL <ISO8601_DURATION>	
Attributes	Signature	Meaning
0..1	range: Interval <ISO8601_DURATION>	Constraint expressed as a range of durations.
0..1	pattern: String	ISO8601-based pattern.
Functions	Signature	Meaning
(redefined)	prototype_value: ISO8601_DURATION	A generated prototype value from this constraint object.
	years_allowed: Boolean	True if years are allowed in the constrained Duration.

CLASS	C_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.
Invariant	<i>Pattern_valid:</i> pattern != Void implies valid_iso8601_duration_constraint_pattern (pattern)	

6.5.22 C_TERMINOLOGY_CODE Class

CLASS	C_TERMINOLOGY_CODE	
Purpose	Constrainer for TERMINOLOGY_CODE type. An instance of this class can represent a coded term constraint as one of the following: <ul style="list-style-type: none"> • <i>constraint</i> = a single at-code • <i>constraint</i> = a single ac-code • <i>constraint</i> = a single ac-code and <i>assumed_value</i> = at-code 	
Inherit	C_PRIMITIVE_OBJECT	
Attributes	Signature	Meaning
(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, <i>assumed_value</i> may contain an at-code from the value set of the ac-code.
Functions	Signature	Meaning
	value_set_expanded: List <String>	Member at-code(s) from archetype-local value set if <i>constraint</i> contains an ac-code.

CLASS	C_TERMINOLOGY_CODE	
	value_set_substituted: List <URI>	List of external URI(s) either substituted for local at-codes in <i>value_set_expanded</i> , or else an external ref-set URI substituted for the ac-code in <i>value_set_id</i> , 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 <i>value_set_substituted</i> . For external value-sets, the list is determined by a terminology service.
(redefined)	prototype_value: TERMINOLOGY_CODE	A generated prototype value from this constraint object.
Invariant	<i>Terminology_id_validity</i> : terminology_id != Void <i>Code_list_validity</i> : code_list != Void	

6.5.23 TERMINOLOGY_CODE Class

CLASS	TERMINOLOGY_CODE	
Purpose	Simple coded term abstraction sufficient for AOM reasoning.	
Inherit	Ordered	
Attributes	Signature	Meaning
1	terminology_id: String	Terminology identifier.
0..1	terminology_version: String	Terminology version identifier.
1	code_string: String	Code or code-phrase from terminology.
Invariant	<i>Terminology_id_validity</i> : terminology_id != Void <i>Code_string_validity</i> : code_string != Void	

6.5.24 C_SECOND_ORDER Class

CLASS	C_SECOND_ORDER (<i>abstract</i>)	
Purpose	Abstract parent of classes defining second order constraints.	
Functions	Signature	Meaning
	members: List <ARCHETYPE_CONSTRAINT>	Members of this second order constrainer. Normally redefined in descendants.

CLASS	C_SECOND_ORDER (abstract)
Invariant	

6.5.25 C_ATTRIBUTE_TUPLE Class

CLASS	C_ATTRIBUTE_TUPLE	
Purpose	Type that represents a tuple constraint on 2 or more sibling C_ATTRIBUTES under a C_COMPLEX_OBJECT.	
Inherit	C_SECOND_ORDER	
Attributes	Signature	Meaning
(redefine)	members: List<C_ATTRIBUTE>	List of C_ATTRIBUTES forming the definition of the tuple.
Functions	Signature	Meaning
	tuple_count: Integer <i>require</i> is_second_order_constrained	Generate size of constrained tuples by inspecting first C_ATTRIBUTE member, and then in turn its first C_PRIMITIVE_OBJECT child, and calling its <i>list_count</i> feature.
Invariant		

7 The Assertion Package

7.1 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 section 8 on page 88. The assertion package is illustrated below in FIGURE 24.

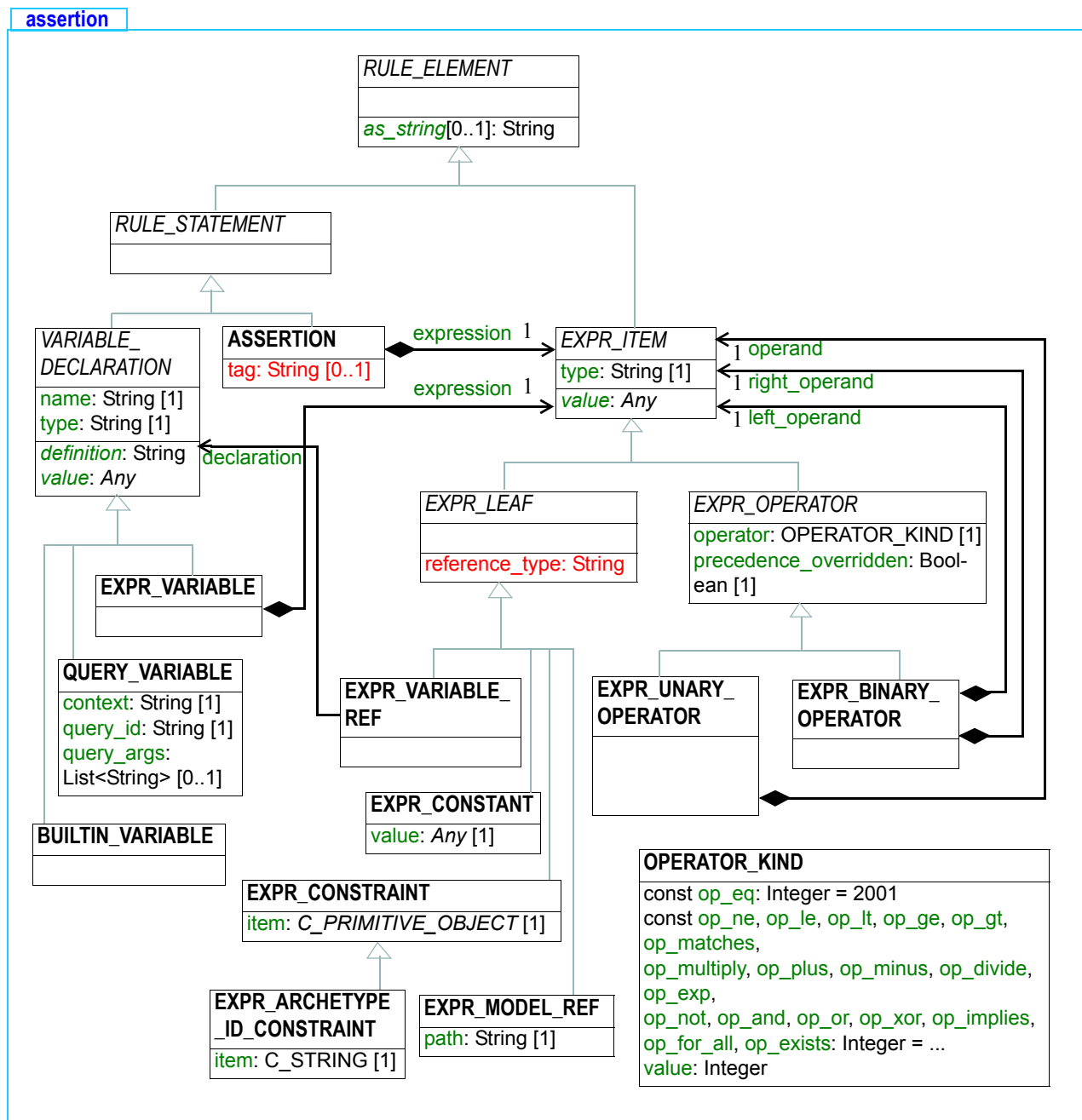


FIGURE 24 The openehr.am.archetype.assertion package

7.2 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.

7.3 Class Descriptions

7.3.1 RULE_STATEMENT Class

CLASS	RULE_STATEMENT (abstract)	
Purpose	Abstract concept of any statement in a block of rule statements.	
Abstract	Signature	Meaning
	as_string : String	Serialised to ADL string form.
Invariant		

7.3.2 ASSERTION Class

CLASS	ASSERTION	
Purpose	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 : String	Expression tag, used for distinguishing multiple assertions.
1	expression : EXPR_ITEM	Root of expression tree.
Invariant	<i>Tag_valid</i> : tag != Void implies not tag.is_empty <i>Expression_valid</i> : expression != Void and then expression.type.is_equal("BOOLEAN")	

7.3.3 VARIABLE_DECLARATION Class

CLASS	VARIABLE_DECLARATION (abstract)	
Purpose	Definition of a named variable used in an assertion expression.	
Inherit	RULE_STATEMENT	
Abstract	Signature	Meaning
	<i>definition</i> : String	Formal definition of the variable.
	<i>value</i> : Any	Value of the variable once evaluated.
Attributes	Signature	Meaning
1	name : String	Name of variable.
1	type : String	Type of variable, from the <i>openEHR</i> assumed types or the <i>openEHR</i> reference model.
Invariant	<i>Name_valid</i> : name != Void and then not name.is_empty <i>Type_valid</i> : type != Void and then not type.is_empty	

7.3.4 EXPR_VARIABLE Class

CLASS	EXPR_VARIABLE	
Purpose	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	expression : EXPR_ITEM	Expression tree of expression.
Invariant	<i>Expression_valid</i> : expression != Void	

7.3.5 BUILTIN_VARIABLE Class

CLASS	BUILTIN_VARIABLE	
Purpose	<p>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 variables is as follows:</p> <ul style="list-style-type: none"> • <code>current_date</code>: ISO8601_DATE • <code>current_time</code>: ISO8601_TIME • <code>current_date_time</code>: ISO8601_DATE_TIME 	
Inherit	VARIABLE_DECLARATION	
Attributes	Signature	Meaning
Invariant		

7.3.6 QUERY_VARIABLE Class

CLASS	QUERY_VARIABLE	
Purpose	<p>Definition of a variable whose value is derived from a query run on a data context 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.</p>	
Inherit	VARIABLE_DECLARATION	
Attributes	Signature	Meaning
0..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	query_id : String	Identifier of query in the external context, e.g. “date_of_birth”. Not yet standardised.
1	query_args : List<String>	Optional arguments to query. Not yet standardised.
Invariant	<p><i>Context_valid</i>: context != Void implies not context.is_empty <i>Query_id_valid</i>: query_id != Void and then not query_id.is_empty</p>	

7.3.7 **EXPR_ITEM Class**

CLASS	EXPR_ITEM (abstract)	
Purpose	Abstract parent of all expression tree items.	
Attributes	Signature	Meaning
1	type: String	Type name of this item in the mathematical sense. For leaf nodes, must be the name of a primitive type, or else a reference model type. The type for any relational or boolean operator will be “Boolean”, while the type for any arithmetic operator, will be “Real” or “Integer”.
Invariant	<i>Type_valid:</i> type != Void and then not type.is_empty	

7.3.8 **EXPR_LEAF Class**

CLASS	EXPR_LEAF	
Purpose	Expression tree leaf item representing one of: <ul style="list-style-type: none"> • 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.
Invariant		

7.3.9 EXPR_CONSTANT Class

CLASS	EXPR_CONSTANT	
Purpose	<p>Constant expression tree leaf item. This can represent a manifest constant of any primitive type, i.e.:</p> <ul style="list-style-type: none"> • 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	value: Any	The constant value.
Invariant	<i>Value_valid:</i> value != Void	

7.3.10 EXPR_CONSTRAINT Class

CLASS	EXPR_CONSTRAINT	
Purpose	<p>Expression tree leaf item representing a constraint on a primitive type, expressed in the form of concrete subtype of C_PRIMITIVE_OBJECT.</p>	
Inherit	EXPR_LEAF	
Attributes	Signature	Meaning
1	item: C_PRIMITIVE_OBJECT	The constraint.
Invariant	<i>Item_valid:</i> item != Void	

7.3.11 EXPR_ARCHETYPE_ID_CONSTRAINT Class

CLASS	EXPR_ARCHETYPE_ID_CONSTRAINT
Purpose	<p>Expression tree leaf item representing a constraint on an archetype identifier.</p>

CLASS	EXPR_ARCHETYPE_ID_CONSTRAINT	
Inherit	EXPR_LEAF	
Attributes	Signature	Meaning
1	item: C_STRING	A constraint on ARCHETYPE_ID objects for use within ARCHETYPE_SLOTS.
Invariant	<i>Constraint_validity</i> : item.is_pattern -- and item.pattern matches ARCHETYPE_ID.pattern_template	

7.3.12 EXPR_MODEL_REF Class

CLASS	EXPR_MODEL_REF	
Purpose	<p>Expression tree leaf item representing a reference to a value found in data at a location specified by a path in the archetype definition.</p> <ul style="list-style-type: none"> 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. “archetype_id/value” refers to the String value of the <i>archetype_id</i> attribute of the enclosing archetype. 	
Inherit	EXPR_ITEM	
Attributes	Signature	Meaning
1	path: String	The path.
Invariant	<i>Path_valid</i> : path != Void	

7.3.13 EXPR_VARIABLE_REF Class

CLASS	EXPR_VARIABLE_REF	
Purpose	Expression tree leaf item representing a reference to a defined variable.	
Inherit	EXPR_LEAF	
Attributes	Signature	Meaning
1	declaration: VARIABLE_DECLARATION	The variable referred to.
Invariant	<i>Declaration_valid</i> : declaration != Void	

7.3.14 EXPR_OPERATOR Class

CLASS	EXPR_OPERATOR (abstract)	
Purpose	Abstract parent of operator types.	
Inherit	EXPR_ITEM	
Attributes	Signature	Meaning
1	operator: OPERATOR_KIND	Code of operator.
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.
Invariant		

7.3.15 EXPR_UNARY_OPERATOR Class

CLASS	EXPR_UNARY_OPERATOR	
Purpose	Unary operator expression node.	
Inherit	EXPR_OPERATOR	
Attributes	Signature	Meaning
1	operand: EXPR_ITEM	Operand node.
Invariant	<i>operand_valid:</i> operand != Void	

7.3.16 EXPR_BINARY_OPERATOR Class

CLASS	EXPR_BINARY_OPERATOR	
Purpose	Binary operator expression node.	
Inherit	EXPR_OPERATOR	
Attributes	Signature	Meaning
1	left_operand: EXPR_ITEM	Left operand node.
1	right_operand: EXPR_ITEM	Right operand node.

CLASS	EXPR_BINARY_OPERATOR
Invariant	<i>left_operand_valid</i> : operand /= Void <i>right_operand_valid</i> : operand /= Void

7.3.17 OPERATOR_KIND Class

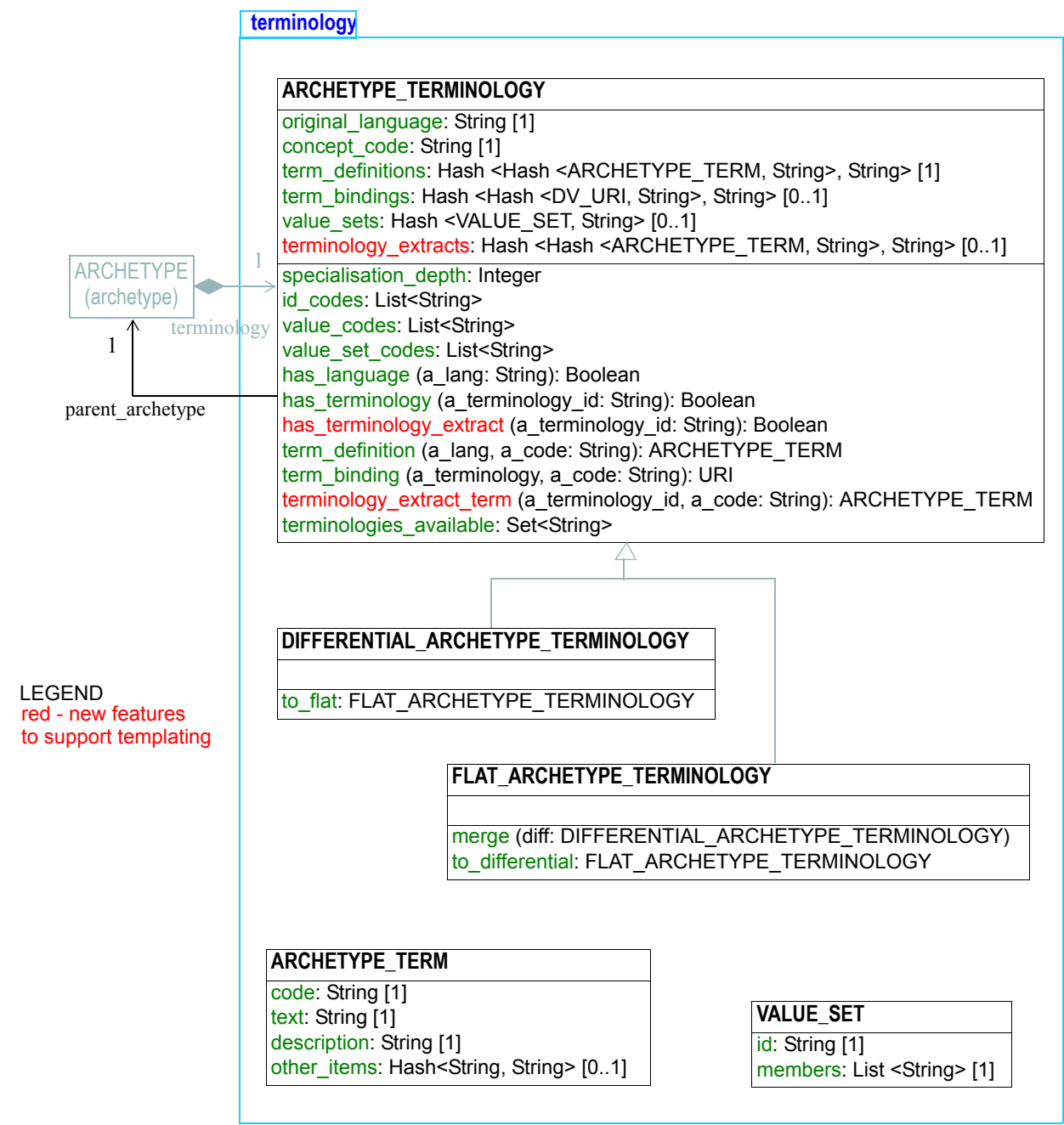
CLASS	OPERATOR_KIND	
Purpose	Enumeration type for operator types in assertion expressions	
Use	Use as the type of operators in the Assertion package, or for related uses.	
Constants	Signature	Meaning
	op_eq : Integer = 2001	Equals operator ('=' or '==')
	op_ne : Integer = 2002	Not equals operator ('!=', '/=' or '<>')
	op_le : Integer = 2003	Less-than or equals operator ('<=')
	op_lt : Integer = 2004	Less-than operator ('<')
	op_ge : Integer = 2005	Greater-than or equals operator ('>=')
	op_gt : Integer = 2006	Greater-than operator ('>')
	op_matches : Integer = 2007	Matches operator ('matches' or 'is_in')
	op_not : Integer = 2010	Not logical operator
	op_and : Integer = 2011	And logical operator
	op_or : Integer = 2012	Or logical operator
	op_xor : Integer = 2013	Xor logical operator
	op_implies : Integer = 2014	Implies logical operator
	op_for_all : Integer = 2015	For-all quantifier operator
	op_exists : Integer = 2016	Exists quantifier operator
	op_plus : Integer = 2020	Plus operator ('+')
	op_minus : Integer = 2021	Minus operator ('-')
	op_multiply : Integer = 2022	Multiply operator ('*')
	op_divide : Integer = 2023	Divide operator ('/')

CLASS	OPERATOR_KIND	
	op_exp : Integer = 2024	Exponent operator ('^')
Attributes	Signature	Meaning
	value : Integer	Actual value of this instance
Functions	Signature	Meaning
	valid_operator (an_op: Integer) : Boolean <i>ensure</i> an_op >= op_eq and an_op <= op_exp	Function to test operator values.
Invariant	<i>Validity</i> : valid_operator(value)	

8 Terminology Package

8.1 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 26.

8.2 Semantics

8.2.1 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;

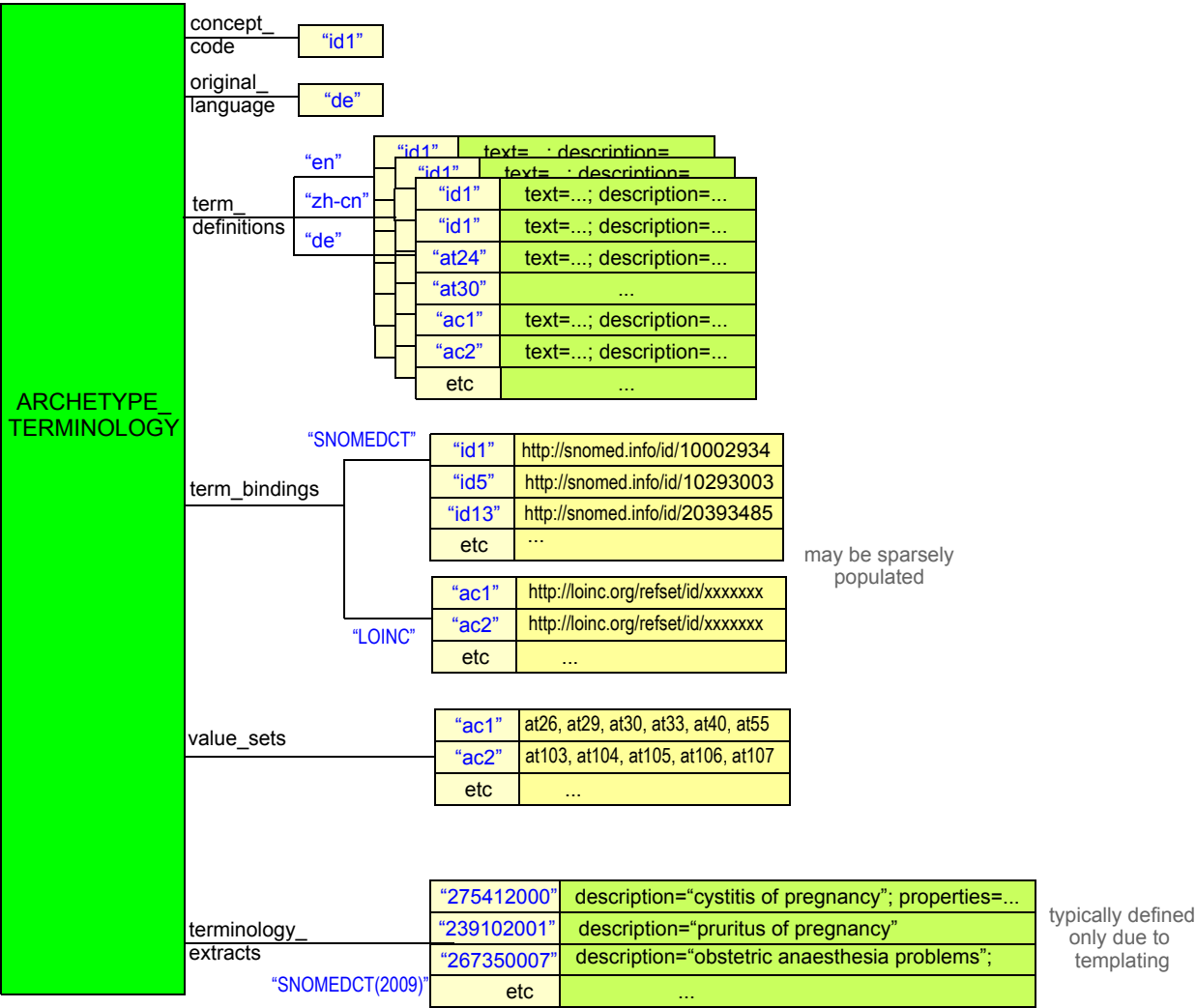


FIGURE 26 Archetype terminology structure.

- '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.

8.3 Class Descriptions

8.3.1 ARCHETYPE_TERMINOLOGY Class

CLASS	ARCHETYPE_TERMINOLOGY
Purpose	Local terminology of an archetype. This class defines the semantics of the terminology of an archetype.

CLASS	ARCHETYPE_TERMINOLOGY	
Attributes	Signature	Meaning
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 <i>term_definitions</i> property.
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	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 <DV_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.
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".
1 (non-persistent)	parent_archetype : ARCHETYPE	Archetype which owns this terminology.
Functions	Signature	Meaning
	id_codes : List<String>	List of all id codes in the terminology, i.e. the "id" codes in an ADL archetype, which are the <i>node_ids</i> 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.

CLASS	ARCHETYPE_TERMINOLOGY	
	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.
	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.
	term_definition (a_lang, a_code: String): ARCHETYPE_TERM <i>require</i> has_term_definition (a_lang, a_code)	Term definition for a code, in a specified language.
	term_binding (a_terminology_id, a_code: String): String <i>require</i> has_term_binding (a_terminology_id), a_code)	Binding of constraint corresponding to <i>a_code</i> in target external terminology <i>a_terminology_id</i> , as a string, which is usually a formal query expression.
	terminology_extract_term (a_terminology_id, a_code: String): ARCHETYPE_TERM <i>require</i> 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_language (a_lang: String): Boolean <i>require</i> a_lang != Void	True if language ‘a_lang’ is present in archetype terminology.
	has_terminology (a_terminology_id: String): Boolean <i>require</i> a_terminology_id != Void	True if terminology <i>a_terminology</i> is present in archetype terminology.
	has_terminology_extract (a_terminology_id: String): Boolean <i>require</i> a_terminology_id != Void	True if there is a terminology extract for <i>a_terminology</i> is present in archetype terminology.

CLASS	ARCHETYPE_TERMINOLOGY	
	terminologies_available: Set<String> <i>ensure</i> Result /= Void	List of terminologies to which term or constraint bindings exist in this terminology, computed from <i>bindings</i> .
	languages_available: Set<String> <i>ensure</i> Result /= Void	List of languages in which terms in this terminology are available.
Invariant	<i>original_language_validity</i> : original_language /= void and then code_set (Code_set_id_languages).has_concept_id (original_language) <i>concept_code_validity</i> : concept_code /= void and then id_codes.has (concept_code) <i>term_definitions_validity</i> term_definitions /= void <i>term_bindings_validity</i> : bindings /= void implies not bindings.is_empty <i>Parent_archetype_valid</i> : parent_archetype /= Void and then parent_archetype.terminology = Current	

8.3.1.1 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.

VTBTK: 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.

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

8.3.2 ARCHETYPE_TERM Class

CLASS	ARCHETYPE_TERM
Purpose	Representation of any coded entity (term or constraint) in the archetype terminology.

CLASS	ARCHETYPE_TERM	
Attributes	Signature	Meaning
1	code: String	Code of this term.
1	text: String	Short term text, typically for display.
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.
Functions	Signature	Meaning
	keys: Set<String> ensure Result != Void	List of all keys used in this term.
Invariant	code_valid: code != void and then not code.is_empty text_valid: text != void description_valid: description != void	

8.3.3 VALUE_SET Class

CLASS	VALUE_SET	
Purpose	Representation of a flat value set within the archetype terminology.	
Attributes	Signature	Meaning
1	id: String	Ac-code of this value set.
1	members: List<String>	List of at-codes in this value set.
Invariant	Id_valid: id != void and then is_valid_value_set_code (id) Members_valid: members != void and then not members.is_empty and then members.for_all (a_member: String is_valid_value_code (a_member))	

Appendix A Algorithms

References

Publications

- 1 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. OOPSLA 2002 workshop on behavioural semantics.
Available at <http://www.deepthought.com.au/it/archetypes.html>.
- 2 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. 2000.
Available at <http://www.deepthought.com.au/it/archetypes.html>.
- 3 Beale T, Heard S. The Archetype Definition Language (ADL). See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/language/ADL/REV_HIST.html.
- 4 Heard S, Beale T. Archetype Definitions and Principles. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/principles/REV_HIST.html.
- 5 Heard S, Beale T. *The openEHR Archetype System*. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/system/REV_HIST.html.
- 6 Rector A L. *Clinical Terminology: Why Is It So Hard?* Yearbook of Medical Informatics 2001.
- 7 W3C. *OWL - The Web Ontology Language*.
See <http://www.w3.org/TR/2003/CR-owl-ref-20030818/>.
- 8 Horrocks *et al.* *An OWL Abstract Syntax*.
See <http://www.w3.org/xxxx/>.

Resources

- 9 openEHR. EHR Reference Model. See <http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/top.html>.
- 10 OMG. The Object Constraint Language 2.0. Available at <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>.

END OF DOCUMENT