*open*EHR

Release 1.1

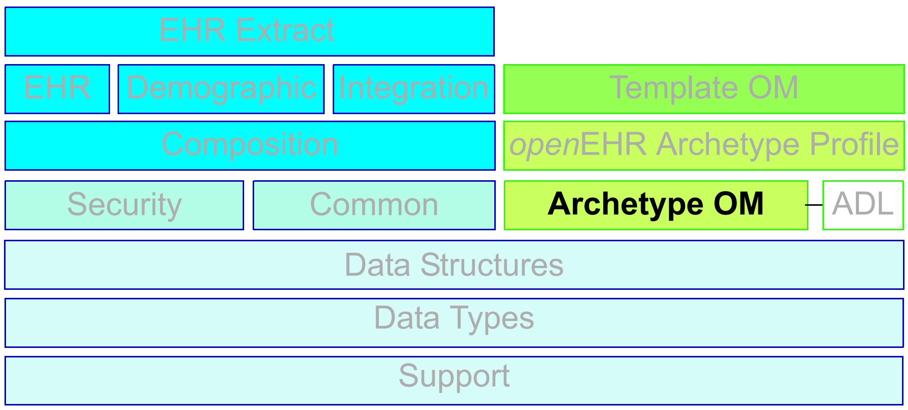
The *open*EHR Archetype Model

Archetype Object Model

Editors: T Beale[a]		
Revision: 2.1	*Pages:* 85	*Date of issue:* 01 Feb 2010
Status: TRIAL		

a. Ocean Informatics

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



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

email: info@openEHR.org **web**: http://www.openEHR.org

Copyright Notice

Amendment Record

Issue	Details	Raiser	Completed
	RELEASE 1.1 candidate		
2.1	**SPEC-270**. Add specialisation semantics to ADL and AOM. Add various attributes and functions to ARCHETYPE_CONSTRAINT descendant classes. • move C_PRIMITIVE.*assumed_value* to attribute slot in UML • rename C_DEFINED_OBJECT.*default_value* function to • correct *assumed_value* definition to be like ; remove its entry from all of the C_PRIMITIVE subtypes • convert BOOLEAN flag representation of patterns to functions and add a String data member for the pattern value, thus matching the XSDs and ADL	T Beale	01 Feb 2010
	SPEC-263. Change Date, Time etc classes in AOM to ISO8601_DATE, ISO8601_TIME etc from Support IM.	T Beale	
	SPEC-296. Convert Interval<Integer> to MULTIPLICITY_INTERVAL to simplify specification and implementation.	T Beale	
	SPEC-300. Archetype slot regular expressions should cover whole identifier. Added C_STRING.*is_pattern*.	A Flinton	
	SPEC-303. Make *existence*, *occurrences* and *cardinality* optional in AOM.	S Heard	
	SPEC-308. Add validity rules to ARCHETYPE_ONTOLOGY.	T Beale	
	SPEC-309. ARCHETYPE_CONSTRAINT adjustments.	T Beale	
	SPEC-311. Add *is_frozen* flag to C_DEFINED_OBJECT to implement cloning.	T Beale	
	SPEC-178. Add template object model to AM. Add *is_exhaustive* attribute to ARCHETYPE_SLOT.	T Beale	
	RELEASE 1.0.2		
2.0.2	**SPEC-257**. Correct minor typos and clarify text. Correct reversed definitions of *is_bag* and *is_set* in CARDINALITY class.	C Ma, R Chen, T Cook	20 Nov 2008
	SPEC-251. Allow both pattern and interval constraint on Duration in Archetypes. Add *pattern* attribute to C_DURATION class.	S Heard	
	RELEASE 1.0.1		
2.0.1	**CR-000200**. Correct Release 1.0 typographical errors. Table for missed class ASSERTION_VARIABLE added. Assumed_value assertions corrected; *standard_representation* function corrected. Added missed *adl_version*, *concept* rename from CR-000153.	D Lloyd, P Pazos, R Chen, C Ma	20 Mar 2007
	CR-000216: Allow mixture of W, D etc in ISO8601 Duration (deviation from standard).	S Heard	
	CR-000219: Use constants instead of literals to refer to terminology in RM.	R Chen	
	CR-000232. Relax validity invariant on CONSTRAINT_REF.	R Chen	
	CR-000233: Define semantics for *occurrences* on ARCHETYPE_INTERNAL_REF.	K Atalag	
	CR-000234: Correct functional semantics of AOM constraint model package.	T Beale	
	CR-000245: Allow term bindings to paths in archetypes.	S Heard	
	RELEASE 1.0		

Issue	Details	Raiser	Completed
2.0	**CR-000153**. Synchronise ADL and AOM attribute naming. **CR-000178**. Add Template Object Model to AM. Text changes only. **CR-000167**. Add AUTHORED_RESOURCE class. Remove description package to resource package in Common IM.	T Beale T Beale T Beale	10 Nov 2005
RELEASE 0.96			
0.6	**CR-000134**. Correct numerous documentation errors in AOM. Including cut and paste error in TRANSLATION_DETAILS class in Archetype package. Corrected hyperlinks in Section 2.3. **CR-000142**. Update ADL grammar to support assumed values. Changed C_PRIMITIVE and C_DOMAIN_TYPE. **CR-000146**: Alterations to am.archetype.description from CEN MetaKnow **CR-000138**. Archetype-level assertions. **CR-000157**. Fix names of OPERATOR_KIND class attributes	D Lloyd S Heard, T Beale D Kalra T Beale T Beale	20 Jun 2005
RELEASE 0.95			
0.5.1	Corrected documentation error - return type of ARCHETYPE_CONSTRAINT.*has_path*; add optionality markers to Primitive types UML diagram. Removed erroneous aggregation marker from ARCHETYPE_ONTOLOGY.*parent_archetype* and ARCHETYPE_DESCRIPTION.*parent_archetype*.	D Lloyd	20 Jan 2005
0.5	**CR-000110**. Update ADL document and create AOM document. Includes detailed input and review from: - DSTC - CHIME, Uuniversity College London - Ocean Informatics Initial Writing. Taken from ADL document 1.2draft B.	T Beale A Goodchild Z Tun T Austin D Kalra N Lea D Lloyd S Heard T Beale	10 Nov 2004

Trademarks

Microsoft is a trademark of the Microsoft Corporation

Acknowledgements

The work reported in this document was funded by Ocean Informatics and University College London (UCL).

Table of Contents

1 Introduction

1.1 Purpose

This document contains the definitive formal statement of archetype semantics, in the form of an object model for archetypes. The model presented here can be used as a basis for building software that processes archetypes, 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 *open*EHR Archetype Definition Language (ADL) [4], XML-instance and so on. As a specification, it can be treated as an API for archetypes.

It is recommended that the *open*EHR 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.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Architecture Overview

Related documents include:

- The *open*EHR Archetype Definition Language (ADL)
- The *open*EHR Archetype Profile (oAP)
- The *open*EHR Template Object Model (TOM)

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'.

1.4 Status

This document is under development, and is published as a proposal for input to standards processes and implementation works.

This document is available at `http://svn.openehr.org/specification/TAGS/Release-1.0.1/publishing/architecture/am/aom.pdf`.

The latest version of this document can be found at `http://svn.openehr.org/specification/TRUNK/publishing/architecture/am/aom.pdf`.

Blue text indicates sections under active development.

1.5 Tools

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

1.6 Changes from Previous Versions

1.6.1 Version 2.0 to 2.1

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.

- The addition of two new classes `DIFFERENTIAL_ARCHETYPE` and `FLAT_ARCHETYPE` which are variants of `ARCHETYPE` class, which is now abstract.
- The addition of two attributes to the `C_ATTRIBUTE` class, allowing the inclusion of a path and a flag including that the matches (∈) operator is to be negated for this attribute. The former allows for specialised archetype redefinitions deep within a structure to be stated with respect to a path rather than having to include the ADL blocks to descend from the top to the point of redefinition. The matches negation flag allows specialised archetypes to state constraints by value exclusion rather than inclusion, which experience has shown is very convenient for some kinds of constraints. All the changes in this version are found in the `constraint_model` and `primitive` packages.
- The `C_DEFINED_OBJECT` *default_value* function has been renamed to , in order to properly represent its meaning (it is a generated value, not a set value) and to avoid a name clash with the *open*EHR Template *default_value* attribute defined in a descendant of the `C_DEFINED_OBJECT` class.
- The addition of two new classes `DIFFERENTIAL_ARCHETYPE_ONTOLOGY` and `FLAT_ARCHETYPE_ONTOLOGY`, which are variants of `ARCHETYPE_ONTOLOGY`, which is now abstract.
- The name of the *invariant* attribute has been changed to *rules*, to better reflect its purpose.

1.6.2 Version 0.6 to 2.0

As part of the changes carried out to ADL version 1.3, the archetype object model specified here is revised, also to version 2.0, to indicate that ADL and the AOM can be regarded as 100% synchronised specifications.

- 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 *open*EHR 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 *open*EHR as well as bindings to terminology.

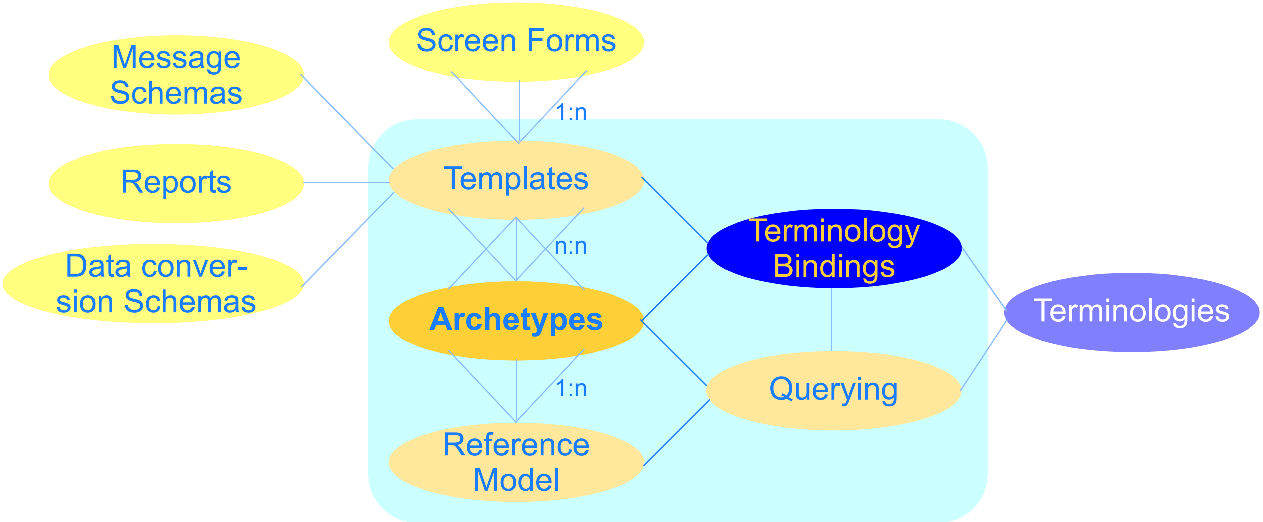


FIGURE 1 The *open*EHR Semantic Architecture

The semantics of archetypes are defined by the following specifications:

- the *open*EHR Archetype Object Model (AOM);
- the *open*EHR Archetype Profile (oAP);
- 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 inform developers how to build software.**

The purpose of the *open*EHR Archetype Profile is to provide custom archetype classes for the *open*EHR reference model.

The *open*EHR archetype framework is described in terms of Archetype Definitions and Principles and *open*EHR Distributed Development Model documents..

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 and the Archetype Profile are used to express the object structures of source archetypes and flattened archetypes. With the addition of a small number of primitives defined in the Template Object Model (TOM), they also express the source and flattened form of *open*EHR templates. 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 *open*EHR reference model class `OBSERVATION` contains the data points relevant to Apgar score of a newborn, while a 'blood pressure measurement' archetype contains 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 environment. 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. 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.

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 *open*EHR ADL specification.

Archetype Composition

It the interests of re-use and clarity of modelling, 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.

2.2.2 Templates

In practical systems, archetypes are assembled into larger usable structures by the use of *open*EHR templates. A template is expressed in a source form similar to that of a specialised archetype, and 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 aretefacts 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.

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. 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.

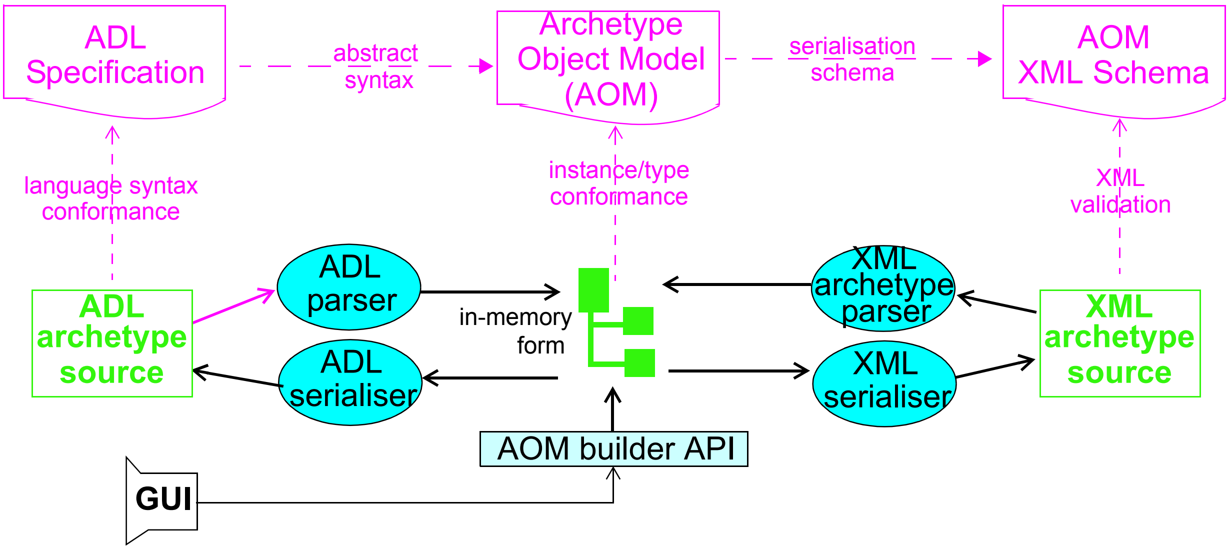


FIGURE 2 Relationship of archetype in-memory and syntax forms

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;

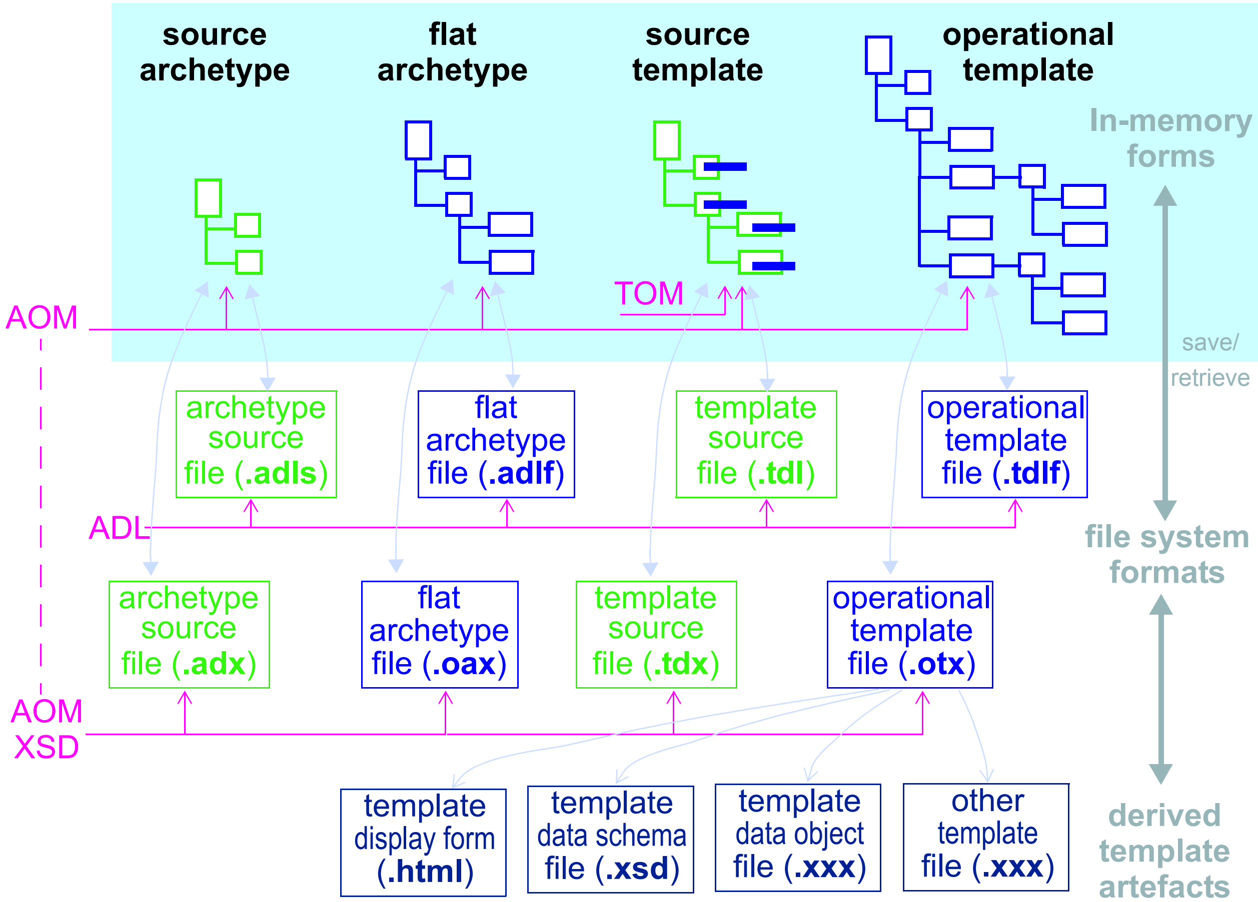


FIGURE 3 Relationship of computational artefacts and specifications

- *template authoring*: creates source-form templates that reference archetypes; also expressed as AOM / TOM 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 lineages* rather than just single archetypes must be processed - i.e. specialised archetypes can only be compiled in conjunction with their specialisation parents up to the top level. 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 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.

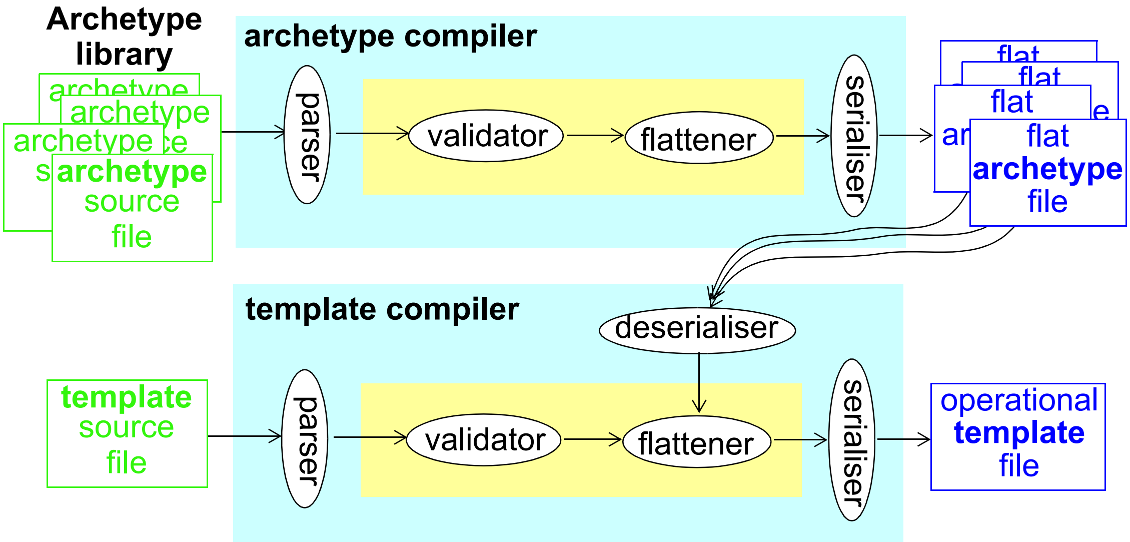


FIGURE 4 Archetype / template tool chain

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 ontology section. It can also validate the classes and attributes mentioned in the archetype against a specification for the relevant reference model[1].

The results of the compilation process can be seen in the archetype visualisations in the *open*EHR ADL Workbench[2].

2.3.4 Optimisations

There is a subtlety in dealing with syntax and in-memory forms of archetypes and templates which becomes important in *open*EHR system design. Artefacts authored by whatever means, including 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 *open*EHR dADL syntax, 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 goal 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. A dADL expression of the *open*EHR reference model is available for this purpose.
2. See http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/apps/doc/adl_workbench_help.htm

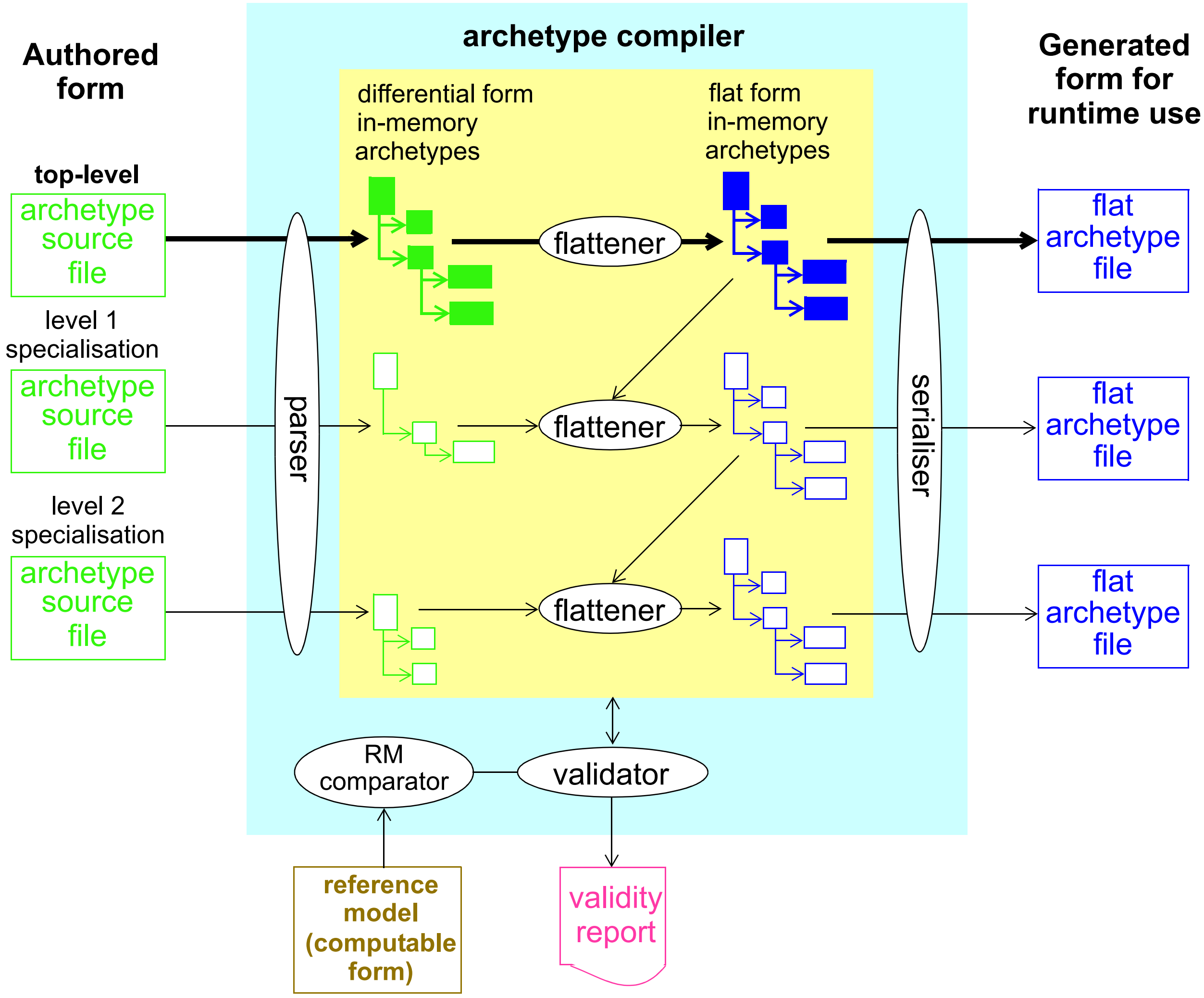


FIGURE 5 Computational model of archetype compilation

Within *open*EHR environments, managing the authoring and persisted forms of archetypes is achieved using various mechanisms including digital signing, which are described in the *open*EHR 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. It is independent of any particular linguistic expression of an archetype, such as ADL or OWL, and can therefore be used with any kind of parser.

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

3.1 Package Structure

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

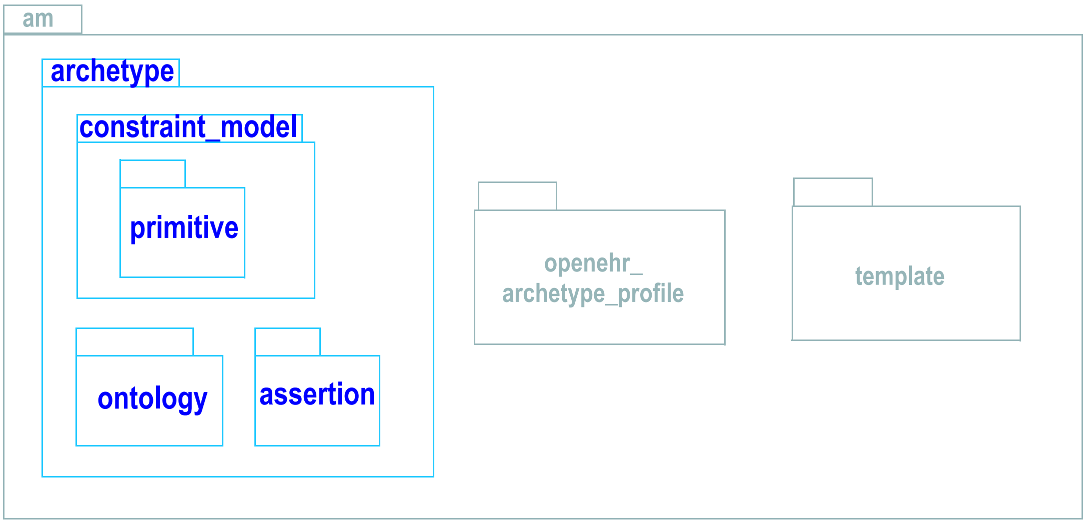


FIGURE 6 openehr.am.archetype Package

4 The Archetype Package

4.1 Overview

The model of an archetype, illustrated in FIGURE 7, is straightforward at an abstract level, mimicking the structure of an archetype document as defined in the *open*EHR Archetype Definition Language (ADL) document. An archetype is a modelled as a particular kind of AUTHORED_RESOURCE, and as such, includes descriptive meta-data, language information and revision history. The ARCHETYPE class adds *identifying information*, a *definition* - expressed in terms of constraints on instances of an object model, optional *rules*, and an *ontology*. The archetype definition, the 'main' part of an arche-

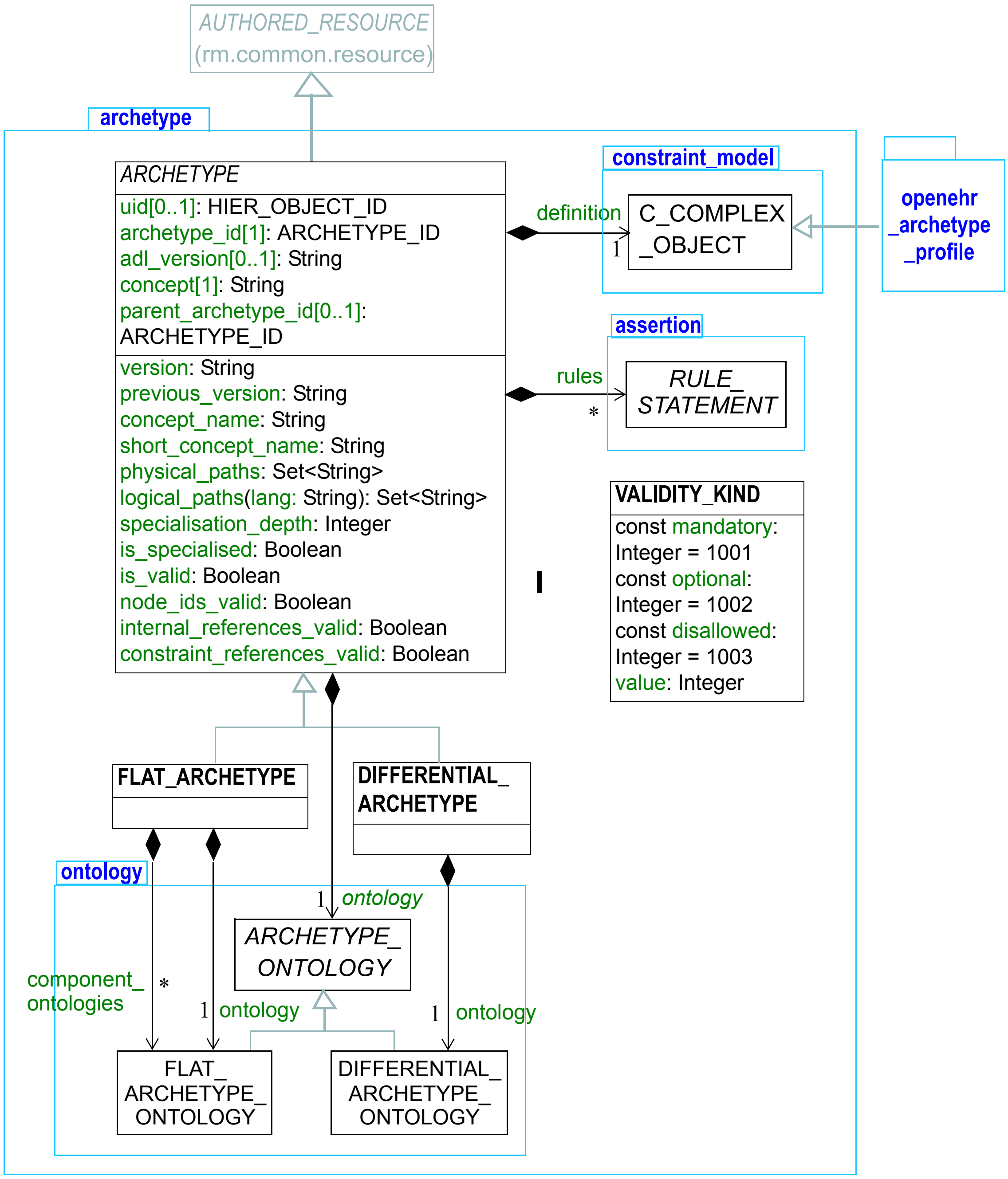


FIGURE 7 openehr.am.archetype Package

type, is an instance of a C_COMPLEX_OBJECT, which is to say, the root of the constraint structure of an archetype always takes the form of a constraint on a non-primitive object type. The last section of an archetype, the ontology, is represented by its own class, and is what allows the archetypes to be natural language- and terminology-neutral.

A utility class, VALIDITY_KIND is also included in the Archetype package. This class contains one integer attribute and three constant definitions, and is intended to be used as the type of any attribute in this constraint model whose value is logically 'mandatory', 'optional', or 'disallowed'. It is used in this model in the classes C_Date, C_Time and C_Date_Time.

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

FIGURE 8 illustrates a object structure common to differential and flat archetypes. Mandatory parts are shown with a bold association.

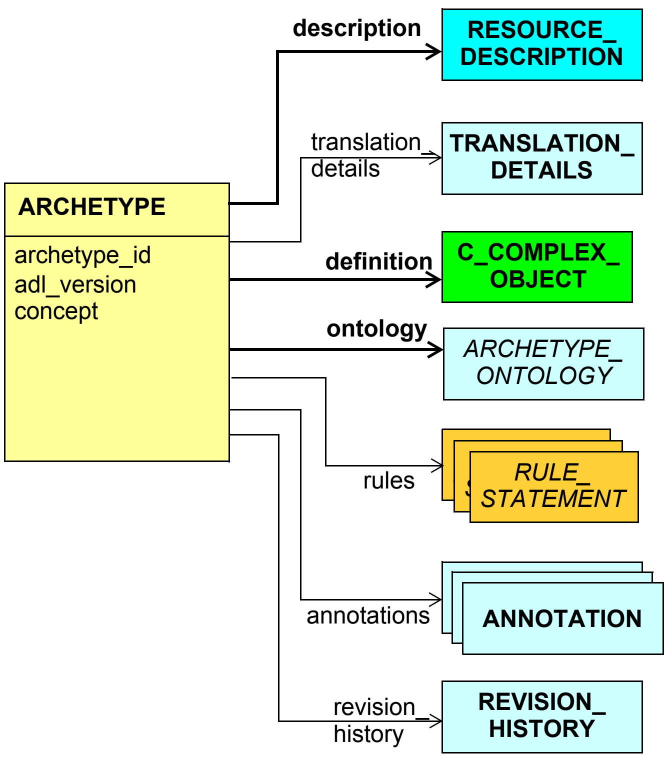


FIGURE 8 Archetype Object Structure

4.2 Class Descriptions

4.2.1 ARCHETYPE Class

CLASS	*ARCHETYPE (abstract)*	
Purpose	Root object of an archetype. Defines semantics of identfication, 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 sharable archetype.
1	**archetype_id**: ARCHETYPE_ID	Multi-axial identifier of this archetype in archetype space.
0..1	**uid**: HIER_OBJECT_ID	OID identifier of this archetype.
1	**concept**: String	The normative meaning of the archetype as a whole, expressed as a local archetype code, typically "at0000".
0..1	**parent_archetype_id**: ARCHETYPE_ID	Identifier of the specialisation parent of this archetype.
1	**definition**: C_COMPLEX_OBJECT	Root node of this archetype
1	**ontology**: ARCHETYPE_ONTOLOGY	The ontology 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	**version**: String	Version of this archetype, extracted from id.
0..1	**previous_version**: String	Version of predecessor archetype of this archetype, if any.
1	**short_concept_name**: String	The short concept name of the archetype extracted from the archetype_id.
	concept_name (a_lang: String): String	The concept name of the archetype in language *a_lang*; corresponds to the term definition of the *concept* attribute in the archetype ontology.

CLASS	ARCHETYPE (abstract)	
1	**physical_paths**: Set<String>	Set of language-independent paths extracted from archetype. Paths obey Xpath-like syntax and are formed from alternations of C_OBJECT.*node_id* and C_ATTRIBUTE.*rm_attribute_name* values.
	logical_paths (a_lang: String): Set<String>	Set of language-dependent paths extracted from archetype. Paths obey the same syntax as physical_paths, but with *node_ids* replaced by their meanings from the ontology.
1	**is_specialised**: Boolean ***ensure*** *Result* **implies** parent_archetype_id /= Void	True if this archetype is a specialisation of another.
1	**specialisation_depth**: Integer ***ensure*** *Result* = ontology. specialisation_depth	Specialisation depth of this archetype; larger than 0 if this archetype has a parent. Derived from *ontology.specialisation_depth*.
	node_ids_valid: Boolean	True if every *node_id* found on a C_OBJECT node is found in *ontology.term_codes*.
	internal_references_valid: Boolean	True if every ARCHETYPE_INTERNAL_REF. *target_path* refers to a legitimate node in the archetype *definition*.
	constraint_references_valid: Boolean	True if every CONSTRAINT_REF.*reference* found on a C_OBJECT node in the archetype *definition* is found in *ontology.constraint_codes*.
	is_valid: Boolean ***ensure*** ***not*** (node_ids_valid **and** internal_references_valid **and** constraint_references_valid) **implies not** *Result*	True if the archetype is valid overall; various tests should be used, including checks on node_ids, internal references, and constraint references.

CLASS	ARCHETYPE (abstract)
Invariant	***archetype_id_validity***: archetype_id /= Void ***concept_valid***: ontology.has_term_code(concept_code) ***uid_validity***: uid /= Void **implies not** uid.is_empty ***version_validity***: version /= Void **and then** version.is_equal(archetype_id.version_id) ***original_language_valid***: original_language /= void **and then** language /= Void ***and then*** code_set(Code_set_id_languages).has_code(original_language) ***description_exists***: description /= Void ***definition_exists***: definition /= Void ***ontology_exists***: ontology /= Void ***Specialisation_validity***: is_specialised **implies** specialisation_depth > 0 ***Rules_valid***: rules /= Void **implies not** rules.is_empty

4.2.1.1 Validity Rules

The following validity rules apply to ARCHETYPE objects in their differential source form:

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

VACSD: archetype concept specialisation depth. the specialisation depth of the concept code must match the specialisation depth of the archetype identifier.

VARDT: archetype definition typename validity. The topmost typename mentioned in the archetype definition section must match the type mentioned in the type-name slot of the first segment of the archetype id.

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.

VACCD: archetype definition code validity. The node identifier of the root node of the definition section must be the concept code mentioned earlier in the archetype.

The following validity rules apply to the description part of the archetype:

VDEOL: original language specified. The description must include an original_language section providing the meta-data of the original authoring language.

The following validity rules apply across the definition and ontology parts of the archetype:

VATDF: archetype term validity. Each archetype term ('at' code) of a given specialisation level used as a node identifier the archetype definition must be defined in the term_definitions part of the ontology of the current archetype or of a specialisation parent, according to specialisation level.

VACDF: constraint code validity. Each constraint code ('ac' code) of a given specialisation level used in the archetype definition part must be defined in the constraint_definitions part of the ontology of the current archetype or of a specialisation parent, according to specialisation level.

VOTM: ontology translations missing. Translations must exist for term_definitions and constraint_definitions sections for all languages defined in the description / translations section.

4.2.2 DIFFERENTIAL_ARCHETYPE Class

CLASS	DIFFERENTIAL_ARCHETYPE	
Purpose	Differential form of an archetype. Also called the 'source' form, as this is the form of an archetype created by an editor. For non-specialised archetypes, this is the same as the flat form. For specialised archetypes, only the differences with respect to the parent are included.	
Inherit	ARCHETYPE	
Attributes	Signature	Meaning
1 (redefined)	**ontology**: DIFFERENTIAL_ARCHETYPE_ONTOLOGY	The differential form ontology of the archetype, which includes only codes and bindings defined in the current archetype.
Invariant		

4.2.3 FLAT_ARCHETYPE Class

CLASS	FLAT_ARCHETYPE	
Purpose	Inheritance-flattened form of an archetype.	
Inherit	ARCHETYPE	
Attributes	Signature	Meaning
1 (redefined)	**ontology**: FLAT_ARCHETYPE_ONTOLOGY	The flat form ontology of the archetype, which includes codes and bindings from all parents.
0..1	**component_ontologies**: Hash <FLAT_ARCHETYPE_ONTOLOGY, String>	Compendium of flattened ontologies of any archetypes externally referenced from this archetype, keyed by archetype identifier. This will almost always be present in a template.
Invariant		

4.2.4 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.

CLASS	VALIDITY_KIND	
Attributes	**Signature**	**Meaning**
1	***const* mandatory**: `Integer = 1001`	Constant to indicate mandatory presence of something
1	***const* optional**: `Integer = 1002`	Constant to indicate optional presence of something
1	***const* disallowed**: `Integer = 1003`	Constant to indicate disallowed presence of something
1	**value**: `Integer`	Actual value
Functions	**Signature**	**Meaning**
	valid_validity (a_validity: `Integer): Boolean` ***ensure*** a_validity >= mandatory **and** a_validity <= disallowed	Function to test validity values.
Invariant	***Validity***: valid_validity(value)	

5 Constraint Model Package

5.1 Overview

FIGURE 9 illustrates 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 UML (or a similar object-oriented meta-model). 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 ontology, 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;
- `ARCHETYPE_INTERNAL_REF`: a node that refers to a previously defined object node in the same archetype. The reference is made using a path;
- `CONSTRAINT_REF`: a node that refers to a constraint on (usually) a text or coded term entity, which appears in the ontology section of the archetype, and in ADL, is referred to with an "acNNNN" code. The constraint is expressed in terms of a query on an external entity, usually a terminology or ontology;
- `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.

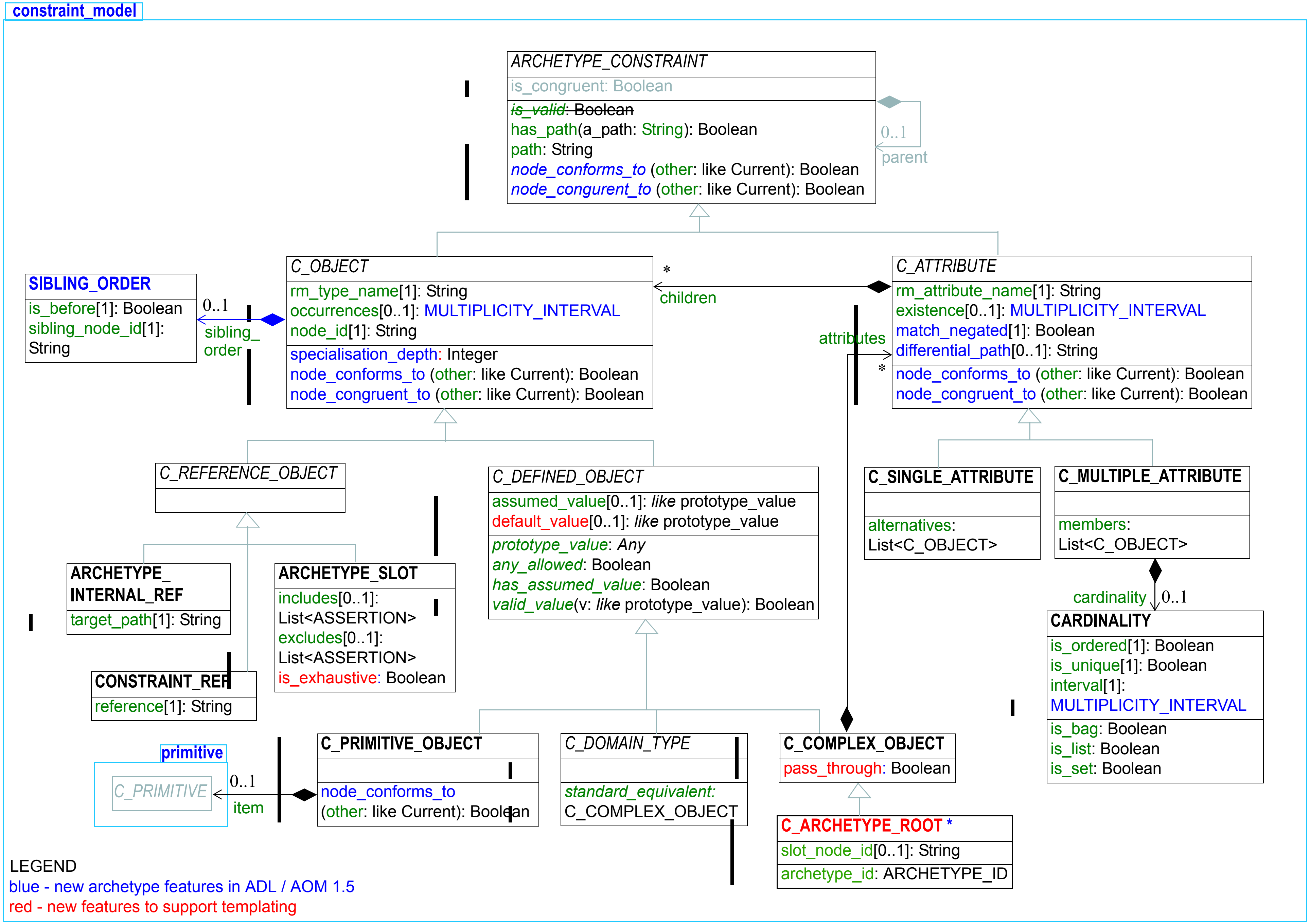


FIGURE 9 openehr.am.archetype.constraint_model Package

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

5.2 Semantics

The effect of the model is to create archetype description structures that are a hierarchical alternation of object and attribute constraints. This structure can be seen by inspecting an ADL archetype, or by viewing an archetype in the *open*EHR 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 subset of the W3C Xpath syntax.

5.2.1 All Node Types

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. The *node_conforms_to* function is used for comparison between corresponding nodes from different archetypes, in order to asert specialisation.

5.2.2 Attribute Node Types

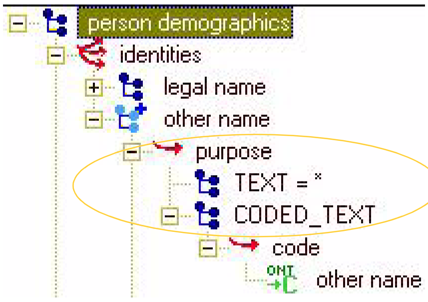
Constraints on attributes are represented by instances of the two subtypes of C_ATTRIBUTE: C_SINGLE_ATTRIBUTE and C_MULTIPLE_ATTRIBUTE. For both subtypes, the common constraint is whether the corresponding instance (defined by the *rm_attribute_name* attribute) must exist. Both subtypes have a list of children, representing constraints on the object value(s) of the attribute.

Single-valued attributes (such as Person.date_of_birth: Date) are constrained by instances of the type C_SINGLE_ATTRIBUTE, which uses the children to represent multiple *alternative* object constraints for the attribute value.

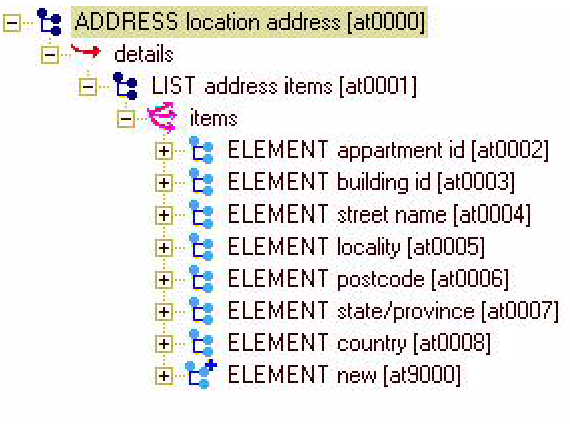
Multiply-valued attributes (such as Person.*contacts*: List<Contact>) are constrained by an instance of C_MULTIPLE_ATTRIBUTE, which allows multiple *co-existing* member objects of the container value of the attribute to be constrained, along with a cardinality constraint, describing ordering and uniqueness of the container. FIGURE 10 illustrates the two possibilities.

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

5.2.3 Object Node Types



C_SINGLE_ATTRIBUTE
with alternatives
(PERSON.address as structured
or free text form)



C_MULTIPLE_ATTRIBUTE
with collection (LIST.items)

FIGURE 10 Single and Multiple-valued C_ATTRIBUTEs

Node_id and Paths

The *node_id* attribute in the class C_OBJECT, inherited by all subtypes, is of great 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 is the main link between the archetype definition (i.e. the constraints) and the archetype ontology, because each *node_id* is a 'term code' in the ontology.

The existence of *node_ids* in an archetype allows archetype paths to be created, which refer to each node. Not every node in the archetype needs a *node_id*, if it does not need to be addressed using a path; any leaf or near-leaf node which has no sibling nodes from the same attribute can safely have no *node_id*.

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_MULTIPLE_ATTRIBUTE for which the cardinality is ordered.

5.2.3.1 Defined Object Nodes (C_DEFINED_OBJECT)

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

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. *Any_allowed* is effected in subtypes to indicate in concrete terms when it is True, usually related to Void attribute values.

Assumed_value

When archetypes are defined to have optional parts, an ability to define ‘assumed’ values is useful. For example, an archetype for the concept ‘blood pressure measurement’ might contain an optional protocol section describing the 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 could be 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 values 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. Assumed values are definable at the leaf level only, which appears to be adequate for all purposes described to date; accordingly, they appear in descendants of `C_PRIMITIVE` and also `C_DOMAIN_TYPE`.

The notion of assumed values is distinct from that of ‘default values’. The latter is a local requirement, and as such is stated in templates; default values *do* appear in data, while assumed values don’t.

Valid_value

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

Prototype_value

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

Default_value

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

5.2.3.2 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_OBJECTs`.

5.2.3.3 Primitive Types

Constraints on primitive types are defined by the classes inheriting from `C_PRIMITIVE`, namely `C_STRING`, `C_INTEGER` and so on. These types do not inherit from `ARCHETYPE_CONSTRAINT`, but rather are related by association, in order to allow them to have the simplest possible definitions, independent even from the rest of ADL, in the hope of acceptance in heath standardisation organisations. Technically, avoiding inheritance from ARCHETYPE_CONSTRAINT / C_PRIMITIVE_OBJECT into these base types (in other words, coalescing the classes C_PRIMITIVE_OBJECT and C_PRIMITIVE) does not pose a problem, but could be effected at a later date if desired.

5.2.3.4 Domain-specific Extensions (C_DOMAIN_TYPE)

The main part of the archetype constraint model allows any type in a reference model to be archetyped - i.e. constrained - in a standard way, which is to say, by a regular cascade of C_COMPLEX_OBJECT / C_ATTRIBUTE / C_PRIMITIVE_OBJECT objects. This generally works well, especially for 'outer' container types in models. However, it occurs reasonably often that lower level logical 'leaf' types need special constraint semantics that are not conveniently achieved with the standard aproach. To enable such classes to be integrated into the generic constraint model, the class C_DOMAIN_TYPE is included. This enables the creation of specific "C_" classes, inheriting from C_DOMAIN_TYPE, which represent custom semantics for particular reference model types. For example, a class called C_QUANTITY might be created which has different constraint semantics from the default effect of a C_COMPLEX_OBJECT / C_ATTRIBUTE cascade representing such constraints in the generic way (i.e. systematically based on the reference model). An example of domain-specific extension classes is shown in Domain-specific Extension Example on page 78.

5.2.3.5 Reference Objects (C_REFERENCE_OBJECT)

The subtypes of C_REFERENCE_OBJECT, namely, ARCHETYPE_SLOT, ARCHETYPE_INTERNAL_REF and CONSTRAINT_REF 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; and a reference to a constraint on a constraint defined in the archetype ontology, which in turn points to an external knowledge resource, such as a terminology.

A CONSTRAINT_REF is really a proxy for a set of constraints on an object that would normally occur at a particular point in the archetype as a C_COMPLEX_OBJECT, but where the actual definition of the constraints is outside the archetype definition proper, and is instead expressed in the binding of the constraint reference (e.g. 'ac0004') to a query or expression into an external service (e.g. a terminology service). The result of the query could be something like:

- a set of allowed CODED_TERMs e.g. the types of hepatitis
- an INTERVAL<QUANTITY> forming a reference range
- a set of units or properties or other numerical item

See the ADL specification for a fuller explanation, under the heading Placeholder constraints in the cADL section.

5.2.4 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 *open*EHR ADL document.

5.3 Class Definitions

5.3.1 ARCHETYPE_CONSTRAINT Class

CLASS	*ARCHETYPE_CONSTRAINT (abstract)*	
Purpose	Archetype equivalent to `LOCATABLE` class in *open*EHR Common reference model. Defines common constraints for any inheritor of `LOCATABLE` in any reference model.	
Abstract	Signature	Meaning
	node_conforms_to (other: **like Current**): `Boolean` ***require*** other /= Void	True if constraints represented by this node are narrower or the same as *other*.
	node_congruent_to (other: **like Current**): `Boolean` ***require*** other /= Void	True if constraints represented by this node contain no redefnitions with respect to the node *other*, with the exception of *node_id* redefnition in `C_OBJECT` nodes.
Attributes	Signature	Meaning
0..1 (non-persistent)	**parent**: `ARCHETYPE_CONSTRAINT`	Parent node in hierarchy. Void if root node.
0..1 (non-persistent)	**is_congruent**: `Boolean`	True if this node is congruent to a corresponding node in a specialisation parent. Only applicable to nodes in specialised, differential archetypes.
Functions	Signature	Meaning
	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 *a_path* exists at this node.
Invariant	***path_exists***: path /= Void	

5.3.2 C_ATTRIBUTE Class

CLASS	*C_ATTRIBUTE(abstract)*
Purpose	Abstract model of constraint on any kind of attribute node.

CLASS	*C_ATTRIBUTE(abstract)*	
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 archteypes.
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 *not* to be matched by the data rather than to be matched.
Functions	**Signature**	**Meaning**
	rm_attribute_path: String	Path of this attribute with respect to owning C_OBJECT, including differential path where applicable.
(redefined)	**path**: String	If *has_differential_path*, returns *rm_attribute_path*, else returns *path* as defined in ARCHETYPE_CONSTRAINT.
(effected)	**node_conforms_to** (other: **like Current**): Boolean ***require*** other /= Void	True if this node on its own (ignoring any subparts) expresses the same or narrower constraints as *other*. Returns False if *cardinality* or *existence* is incompatible.

CLASS	C_ATTRIBUTE(abstract)	
(effected)	**node_congruent_to** (other: **like Current**): `Boolean` ***require*** other /= Void	True if this node on its own (ignoring any subparts) expresses the same constraints as *other*.
	existence_conforms_to (other: **like Current**): `Boolean` ***require*** other /= Void	True if the existence of this node conforms to existence of node *other*; returns True if the existence of this attribute is Void.
	cardinality_conforms_to (other: **like Current**): `Boolean` ***require*** other /= Void	True if the cardinality of this node conforms to cardinality of node *other*, returns True if the cardinality of this attribute is Void.
	has_differential_path: `Boolean`	True if differential_path is not Void..
	occurrences_total_range: `MULTIPLICITY_INTERVAL`	Minimal cardinality interval bounding occurrences of all child object nodes.
Invariant	***Rm_attribute_name_valid***: rm_attribute_name /= Void **and then not** rm_attribute_name.is_empty ***Existence_valid***: existence /= Void **implies** (existence.lower >= 0 **and** existence.upper <= 1) ***Children_validity***: any_allowed **xor** children /= Void ***Children_occurrences_validity***: cardinality /= Void **implies** cardinality.interval.contains (occurrences_total_range) ***Differential_path_valid***: differential_path /= Void **implies not** differential_path.is_empty ***Has_differential_path_valid***: differential_path = Void **xor** has_differential_path	

5.3.2.1 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.

```
node_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := existence_conforms_to (other) and
        ((is_single and other.is_single) or cardinality_conforms_to (other))
    end

node_congruent_to (other: like Current): Boolean
    require
        other /= Void
    do
```

```
        Result := node_conforms_to(other)
    end

existence_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
        other_is_flat: other.existence /= Void
    do
        Result := existence = Void or
                  existence.is_equal (other.existence) or
                  other.existence.contains (existence)
    end

cardinality_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
        other_is_flat: other.cardinality /= Void
    do
        Result := cardinality = Void or
              cardinality.interval.is_equal (other.cardinality.interval) or
              other.cardinality.contains (cardinality)
    end
```

5.3.2.2 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 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.

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, this path must 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.

5.3.3 C_SINGLE_ATTRIBUTE Class

CLASS	C_SINGLE_ATTRIBUTE	
Purpose	Concrete model of constraint on a single-valued attribute node. The meaning of the inherited children attribute is that they are alternatives.	
Functions	**Signature**	**Meaning**
	alternatives: `List<C_OBJECT>`	List of alternative constraints for the single child of this attribute within the data.
Invariant	***Alternatives_valid***: alternatives /= Void **and then** alternatives.for_all(co: `C_OBJECT` | co.occurrences.upper <= 1)	

5.3.3.1 Validity Rules

The following validity rules apply to single-valued attributes:

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.

VACSU: single-valued attribute child node uniqueness: any object node added as a child to a single-valued attribute must either have a node identifier or reference model type that is unique with respect to the node identifier or the reference model type of all other siblings.

VACSI: single-valued attribute child node identifier: any object node with a node identifier added as a child to a single-valued attribute must have a node identifier that is unique with respect to the node identifiers of all other siblings.

VACSIT: single-valued attribute child node reference model type: any object node without a node identifier added as a child to a single-valued attribute must have a reference model type that is unique with respect to the reference model types of all other siblings.

5.3.4 C_MULTIPLE_ATTRIBUTE Class

CLASS	C_MULTIPLE_ATTRIBUTE	
Purpose	Concrete model of constraint on multiply-valued (ie. container) attribute node.	
Attributes	**Signature**	**Meaning**
0..1	**cardinality**: `CARDINALITY`	Cardinality of this attribute constraint, if it constraints a container attribute.
Functions	**Signature**	**Meaning**

CLASS	C_MULTIPLE_ATTRIBUTE	
	members: `List<C_OBJECT>`	List of constraints representing members of the container value of this attribute within the data. Semantics of the uniqueness and ordering of items in the container are given by the *cardinality*.
	occurrences_total_range: `MULTIPLICITY_INTERVAL`	Total range generated from *occurrences* of all members as sum(all occurrences.lower) .. sum(all occurrences.upper). Only valid on flat archetypes.
Invariant	***Child_occurrences_validity***: cardinality /= Void **implies** cardinality.interval.contains(occurrences_total_range)	

5.3.4.1 Validity Rules

The following validity rules apply to container attributes:

VACMI: child node identification: any object node added as a child to a container attribute must have a node identifier.

VACMM: child node identifier uniqueness: the node identifier of an object node added as a child to a container attribute must be unique with respect to the siblings in the container.

VACMC: cardinality/occurrences validity: where occurrences and cardinality are stated, the interval represented by:
(sum of all occurrences minimum values) .. (sum of all occurrences maximum values)
must intersect with the interval stated by the cardinality.

TBD_1: this should probably be relaxed, since if a cardinality is narrowed in a child, we would have to narrow the occurrences of all the children to satisfy this rule.

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 rule applies to cardinality redefinition in a specialised archetype:

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

5.3.5 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.
1	**interval**: `MULTIPLICITY_INTERVAL`	The interval of this cardinality.
Functions	Signature	Meaning
	is_set: `Boolean` ***ensure*** *Result* = **not** is_ordered **and** is_unique	True if the semantics of this cardinality represent a set, i.e. unordered, unique membership.
	is_list: `Boolean` ***ensure*** *Result* = 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` ***ensure*** *Result* = **not** is_ordered **and not** is_unique	True if the semantics of this cardinality represent a bag, i.e. unordered, non-unique membership.
Invariant	***Validity***: **not** interval.lower_unbounded	

5.3.6 C_OBJECT Class

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

CLASS	C_OBJECT (abstract)	
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 underlying reference model or parent archetype in the case of specialised archteypes.
1	**node_id**: String	Semantic id of this node, used to differentiate sibling nodes of the same type. [Previously called 'meaning']. Each *node_id* must be defined in the archetype ontology as a term code.
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_MULTIPLE_ATTRIBUTE.
Functions	**Signature**	**Meaning**
(effected)	**node_conforms_to** (other: ***like* Current**): Boolean ***require*** other /= Void	True if this node on its own (ignoring any subparts) expresses the same or narrower constraints as `other'. Returns False if any of rm_type_name, occurrences, node_id (& specialisation depth) is incompatible. *Note*: not easily evaluatable for CONSTRAINT_REF nodes.
(effected)	**node_congruent_to** (other: ***like* Current**): Boolean ***require*** other /= Void	True if this node on its own (ignoring any subparts) expresses the same constraints as `other'. Returns False if any of rm_type_name, occurrences, sibling order is different:. The node_id may be redefined however.
	rm_type_conforms_to (other: ***like* Current**): Boolean ***require*** other /= Void	True if this node *rm_type_name* conforms to other.*rm_type_name* by either being equal, or by being a subtype, according to the underlying reference model.

CLASS	C_OBJECT (abstract)	
	occurrences_conforms_to (other: **like Current**): `Boolean` ***require*** 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: ***like*** **Current**): `Boolean` ***require*** other /= Void	True if this node id conforms to other.node_id.
	specialisation_depth: `Integer`	Level of specialisation of this archetype node, based on its *node_id*. The value 0 corresponds to non-specialised, 1 to first-level specialisation and so on. The level is the same as the number of '.' characters in the *node_id* code. If *node_id* is not set, the return value is -1, signifying that the specialisation level should be determined from the nearest parent `C_OBJECT` node having a *node_id*.
Invariant	***Rm_type_name_valid***: rm_type_name /= Void **and then not** rm_type_name.is_empty ***Node_id_valid***: node_id /= Void **and then not** node_id.is_empty ***Occurrences_validity***: (occurrences /= Void **and** parent /= Void **and** parent.is_single) **implies** occurrences.upper <= 1 ***Sibling_order_validity***: sibling_order /= Void **implies** specialisation_depth > 0 and parent.is_multiple	

5.3.6.1 Conformance and congruence 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.

```
node_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        if is_addressable and other.is_addressable then
            if node_id.is_equal (other.node_id) then
                Result := rm_type_name.is_equal (other.rm_type_name) and
                          occurrences.is_equal(other.occurrences) -- maybe just
conforms
            else
                Result := (rm_type_conforms_to(other) and
                          occurrences_conforms_to (other) and
                          node_id_conforms_to (other))
            end
        elseif not is_addressable and not other.is_addressable then
            Result := rm_type_conforms_to(other) and
                      occurrences_conforms_to (other)
        end
```

```
    end

node_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.is_equal (other.occurrences) and
                  node_id_conforms_to (other)
    end

rm_type_conforms_to (other: like Current): Boolean
    require
        other /= Void
    do
        Result := rm_type_name.is_equal (other.rm_type_name) or
              rm_checker.is_sub_type_of (rm_type_name, other.rm_type_name)
    end

occurrences_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
        other_is_flat: other.occurrences /= Void
    do
        Result := occurrences = Void or
              occurrences.is_equal (other.occurrences) or
              other.occurrences.contains (occurrences)
    end

node_id_conforms_to (other: like Current): Boolean
    require
        other_exists: other /= Void
    do
        Result := node_id.starts_with (other.node_id)
    end
```

5.3.6.2 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.

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

> **VSONT: specialised archetype object node meta-type conformance**: the meta-type of a redefined object node (i.e. the AOM node type such as C_COMPLEX_OBJECT etc) in a specialised archetype must be the same as that of the corresponding node in the flat parent, with the exceptions of the

ARCHETYPE_INTERNAL_REF and CONSTRAINT_REF meta-types (see validity rules VSUNT and VSCNR).

VSONCT: specialised archetype object node reference type conformance: the reference model type of a redefined object node in a specialised archetype must conform to the reference model type in the corresponding node in the flat parent archetype by either being identical, or conforming via an inheritance relationship in the relevant reference model.

VSONI: specialised archetype object node correspondence: an object node in a specialised archetype must carry a node identifier if the corresponding node in the parent carries an identifier, and may not, if the corresponding parent does not.

VSONIR: specialised archetype object node redefinition: if it exists, the node identifier of an object node in a specialised archetype must be redefined into its specialised form if either reference model type or occurrences of the immediate object constraint is redefined.

VSONCI: specialised archetype object node identifier conformance: if defined, the node identifier of a redefined object node in a specialised archetype must conform to the node identifier in the corresponding node in the flat parent archetype by either being identical, or being a derived identifier at the specialisation level of the child archeytpe.

VSONCO: specialised archetype object node occurrences conformance: 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.

VSSM: specialised archetype sibling order validity: the sibling order node id code used in a sibling marker in a specialised archetype must refer to a node found within the same container in the flat parent archetype.

5.3.7 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.
Invariant	***sibling_node_id_validity***: sibling_node_id /= Void	

5.3.8 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**
	prototype_value: `Any`	Generate a prototype value from this constraint object
	valid_value `(a_value:` **like** `prototype_value): Boolean` ***require*** `a_value /= Void`	True if a_value is valid with respect to constraint expressed in concrete instance of this type.
	any_allowed: `Boolean`	True if any value (i.e. instance) of the reference model type would be allowed. Redefined in descendants.
Attributes	**Signature**	**Meaning**
0..1	**assumed_value:** ***like*** `prototype_value`	Value to be assumed if none sent in data
0..1	**default_value**: ***like*** `prototype_value`	Default value set in a template, and present in an operational template. Generally limited to leaf and near-leaf nodes.
Functions	**Signature**	**Meaning**
	has_assumed_value: `Boolean`	True if there is an assumed value
	has_default_value: `Boolean`	True if there is a default value
Invariant	***Assumed_value_valid***: has_assumed_value **implies** assumed_value.conforms_to_type(rm_type_name) **and** valid_value(assumed_value) ***Default_value_valid***: has_default_value **implies** default_value.conforms_to_type(rm_type_name) **and** valid_value(default_value)	

5.3.8.1 Validity Rules

The validity rules for `C_DEFINED_OBJECTs` are as follows:

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

5.3.9 C_COMPLEX_OBJECT Class

CLASS	C_COMPLEX_OBJECT	
Purpose	Constraint on complex objects, i.e. any object that consists of other object constraints.	
Inherit	C_DEFINED_OBJECT	
Functions	**Signature**	**Meaning**
(effected)	**any_allowed**: Boolean ***ensure*** Result = attributes.is_empty	True if any value of the reference model type being constrained is allowed.
Attributes	**Signature**	**Meaning**
0..1	**pass_through**: Boolean	Set to True to indicate that this node can be omitted in rendering, e.g. to the screen, in a report or other visual context. Tpically applied to nodes where the meaning is repeated the next level down.
0..1	**attributes**: Set<C_ATTRIBUTE>	List of constraints on attributes of the reference model type represented by this object.
Invariant	***attributes_valid***: attributes /= Void	

5.3.9.1 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.

5.3.10 C_ARCHETYPE_ROOT Class

CLASS	C_ARCHETYPE_ROOT	
Purpose	A specialisation of `C_COMPLEX_OBJECT` that is specified via an archetype identifier. This can be used to act as a reference to an archetype within another archetype. Within a template, it is used as the root object and as a slot-filler; in both cases, it can refer to another template as well as an archetype. When used as a slot filler in a template, the *slot_node_id* attribute is set to match the *node_id* of the slot being filled. When used in a source archetype there are no attribute children; when used in a template, any attribute sub-structure is an 'overlay' of the same form as a specialised archetype. In an operational template, the structure contains the result of flattening any template overlay structure and the underlying flat archetype. The only formal difference from a normal `C_COMPLEX_OBJECT` is that the *node_id* attribute is an archetype or template identifier rather than an archetype-internal node-code.	
Inherit	`C_COMPLEX_OBJECT`	
Attributes	**Signature**	**Meaning**
0..1	**slot_node_id**: `String`	Identifier of slot, if this archetype is being used to fill a slot.
Functions	**Signature**	**Meaning**
	archetype_id: `ARCHETYPE_ID` ***ensure*** Result /= Void	Identifier of the archetype generated as an `ARCHETYPE_ID` object from the inherited *node_id* value which contains the string form of the identifier.
Invariant	***Node_id_validity***: archetype_ref.valid_id(node_id) ***Slot_node_id_validity***: slot_node_id /= Void **implies not** slot_node_id.is_empty	

5.3.11 C_PRIMITIVE_OBJECT Class

CLASS	C_PRIMITIVE_OBJECT	
Purpose	Constraint on a primitive type.	
Inherit	`C_DEFINED_OBJECT`	
Functions	**Signature**	**Meaning**
(effected)	**any_allowed**: `Boolean` ***ensure*** *Result* = (item = Void)	True if any value of the type being constrained in *item* is allowed.

CLASS	C_PRIMITIVE_OBJECT	
(redefined)	**node_conforms_to** (other: **like Current**): `Boolean` ***ensure*** *Result* = precursor(other) **and** (other.any_allowed **or** (**not** any_allowed **and** item.node_conforms_to (other.item)	True if this node is a subset of, or the same as `other'.
Attributes	**Signature**	**Meaning**
0..1	**item**: `C_PRIMITIVE`	Object actually defining the constraint.
Invariant	***item_exists***: any_allowed **xor** item /= Void	

5.3.12 C_DOMAIN_TYPE Class

CLASS	*C_DOMAIN_TYPE (abstract)*	
Purpose	Abstract parent type of domain-specific constrainer types, to be defined in external packages.	
Inherit	`C_DEFINED_OBJECT`	
Abstract	**Signature**	**Meaning**
	standard_equivalent: `C_COMPLEX_OBJECT`	Standard (i.e. `C_OBJECT`) form of constraint.
Invariant		

5.3.13 C_REFERENCE_OBJECT Class

CLASS	*C_REFERENCE_OBJECT (abstract)*	
Purpose	Abstract parent type of `C_OBJECT` subtypes that are defined by reference.	
Inherit	`C_OBJECT`	
Abstract	**Signature**	**Meaning**
Invariant		

5.3.14 ARCHETYPE_SLOT Class

CLASS	**ARCHETYPE_SLOT**	
Purpose	Constraint describing a 'slot' where another archetype can occur.	
Inherit	`C_REFERENCE_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_exhaustive**: `Boolean`	True if this slot specification in this template exhaustively mentions all fillers, in which case, the slot will not be available for further filling at runtime. Default value False, i.e. unless explicitly set, slots remain open.
Invariant	***includes_valid***: includes /= Void **implies not** includes.is_empty ***excludes_valid***: excludes /= Void **implies not** excludes.is_empty ***validity***: any_allowed **xor** (includes /= Void **or** excludes /= Void)	

5.3.14.1 Validity Rules

The validity rules for `ARCHETYPE_SLOT`s 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 *open*EHR specification for archetype identifiers.

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 open (is_exhaustive = False).

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

5.3.15 ARCHETYPE_INTERNAL_REF Class

CLASS	ARCHETYPE_INTERNAL_REF	
Purpose	A constraint defined by proxy, using a reference to an object constraint defined elsewhere in the same archetype. Note that since this object refers to another node, there are two objects with available occurrences values. The local *occurrences* value on an `ARCHETYPE_INTERNAL_REF` should always be used where set. When setting this from a serialised form, if no occurrences is mentioned, the target occurrences should be used (not the standard default of {1..1}); otherwise the locally specified occurrences should be used as normal. When serialising out, if the occurrences is the same as that of the target, it can be left out.	
Inherit	`C_REFERENCE_OBJECT`	
Attributes	**Signature**	**Meaning**
1	**target_path**: `String`	Reference to an object node using archetype path notation.
Invariant	***Consistency:*** **not** any_allowed ***target_path_valid***: target_path /= Void **and then not** target_path.is_empty -- **and then** ultimate_root.has_path(target_path)	

5.3.15.1 Validity Rules

The following validity rules applies to internal references:

VUNT: use_node reference model type validity: the reference model type mentioned in an ARCHETYPE_INTERNAL_REF node must be the same as or a supertype (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 meta-type validity: a ARCHETYPE_INTERNAL_REF node may be redefined in a specialised archetype by another ARCHETYPE_INTERNAL_REF (e.g. in order to redefine occurrences), or by a node structure that legally redefines the node referred to by the reference, according to other validity rules.

5.3.16 ARCHETYPE_EXTERNAL_REF Class

CLASS	ARCHETYPE_EXTERNAL_REF
Purpose	A constraint object that allows a direct reference to another archteype. This is generally used to refer to lower level reusable archetypes.

CLASS	ARCHETYPE_EXTERNAL_REF	
Inherit	C_REFERENCE_OBJECT	
Attributes	**Signature**	**Meaning**
1	**reference**: ARCHETYPE_ID	Reference to another archetype.
Invariant	***Consistency:*** **not** any_allowed ***Reference_valid***: reference /= Void	

5.3.16.1 Validity Rules

The following validity rules apply to external references:

> **VXRE: external reference exists**: the archetype identifier must exist in the system of archetypes in which the current archetype is validated.
>
> **VXRT: external reference type validity**: the archetype referred to must be of a reference model type from the .same reference model as the current archetype, and the type must be conformant to the type expected at the position it appears.

5.3.17 CONSTRAINT_REF Class

CLASS	CONSTRAINT_REF	
Purpose	Reference to a constraint described in the same archetype, but outside the main constraint structure. This is used to refer to constraints expressed in terms of external resources, such as constraints on terminology value sets.	
Inherit	C_REFERENCE_OBJECT	
Attributes	**Signature**	**Meaning**
1	**reference**: String	Reference to a constraint in the archetype local ontology.
Invariant	***Consistency:*** **not** any_allowed ***reference_valid***: reference /= Void	

5.3.17.1 Validity Rules

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

> **VSCNR: placeholder constraint node conformance**: a placeholder node can only be defined into a reference model type conformant with the type of the original constraint in the parent archetype.

6 The Primitive Package

6.1 Overview

Ultimately any archetype definition will devolve down to leaf node constraints on instances of primitive types. The primitive package, illustrated in FIGURE 11, defines the semantics of constraint on such types.

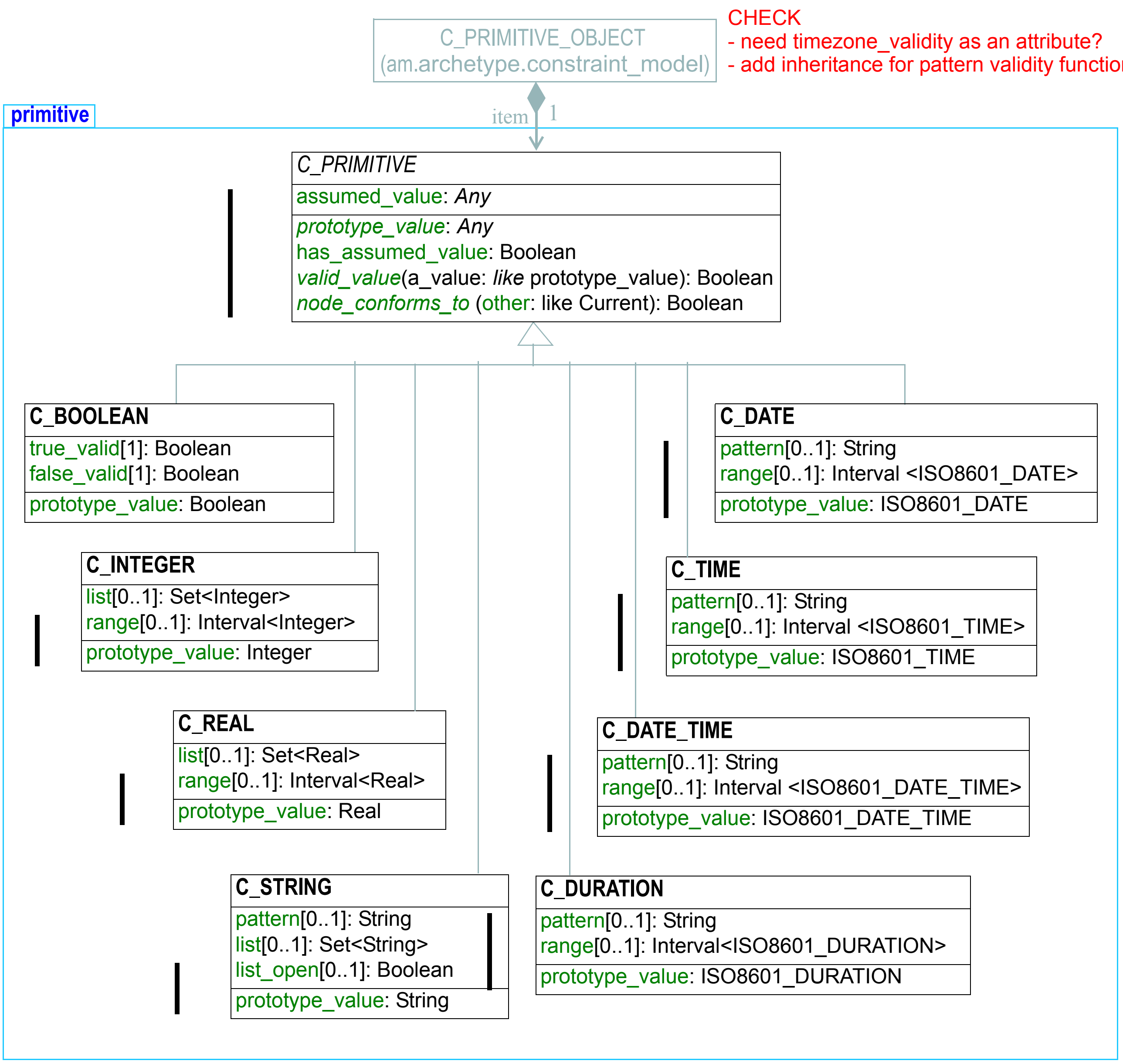


FIGURE 11 The openehr.am.archetype.primitive Package

Most of the types provide at least two alternative ways to represent the constraint; for example the `C_DATE` type allows the constraint to be expressed in the form of a pattern (defined in the ADL specification) or an `Interval<Date>`. Note that the interval form of dates is probably only useful for historical date checking (e.g. the date of an antique or a particular batch of vaccine), rather than constraints on future date/times.

6.2 Class Descriptions

6.2.1 C_PRIMITIVE Class

CLASS	*C_PRIMITIVE (abstract)*	
Purpose	Abstract supertype of all prmitive types.	
Attributes	**Signature**	**Meaning**
0..1	**assumed_value**: ***like*** prototype_value	Value to be assumed if none sent in data.
Functions	**Signature**	**Meaning**
	prototype_value: `Any`	A generated prototype value from this constraint object. Redefined in all descendants.
	valid_value `(a_value:` **like** prototype_value) : `Boolean` ***require*** `a_value /= Void`	True if a_value is valid with respect to constraint expressed in concrete instance of this type.
	node_conforms_to (other: **like Current**): `Boolean` ***require*** other /= Void	True if this node is a subset of, or the same as `other'.
	has_assumed_value: `Boolean` ***ensure*** Result = assumed_value /= Void	True if there is an assumed value.
Invariant	***Assumed_value_valid***: has_assumed_value **implies** valid_value(assumed_value)	

6.2.2 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`	
Attributes	**Signature**	**Meaning**
1	**true_valid**: `Boolean`	True if the value True is allowed
1	**false_valid**: `Boolean`	True if the value False is allowed

CLASS	C_BOOLEAN	
Functions	**Signature**	**Meaning**
(redefined)	**prototype_value**: `Boolean`	A generated prototype value from this constraint object.
Invariant	***Binary_consistency***: true_valid ***or*** false_valid ***Prototype_value_consistency***: .value ***and*** true_valid ***or else not*** .value ***and*** false_valid	

6.2.3 C_STRING Class

CLASS	C_STRING	
Purpose	Constraint on instances of `STRING`.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**pattern**: `String`	Regular expression pattern for proposed instances of `String` to match.
0..1 (cond)	**list**: `Set<String>`	Set of Strings specifying constraint
1	**list_open**: `Boolean`	True if the list is being used to specify the constraint but is not considered exhaustive.
Functions	**Signature**	**Meaning**
(redefined)	**prototype_value**: `String`	A generated prototype value from this constraint object.
Functions	**Signature**	**Meaning**
	is_pattern: `Boolean`	True if *pattern* is not Void.
Invariant	***Consistency***: is_pattern **xor** list /= Void ***Pattern_validity***: is_pattern **implies not** pattern.is_empty ***List_open_validity***: list_open **implies not** is_pattern	

6.2.4 C_INTEGER Class

CLASS	C_INTEGER
Purpose	Constraint on instances of `Integer`.
Inherit	`C_PRIMITIVE`

CLASS	C_INTEGER	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**list**: `Set<Integer>`	Set of Integers specifying constraint
0..1 (cond)	**range**: `Interval<Integer>`	Range of Integers specifying constraint
Functions	**Signature**	**Meaning**
(redefined)	**prototype_value**: `Integer`	A generated prototype value from this constraint object.
Invariant	***Consistency***: list /= Void ***xor*** range /= Void	

6.2.5 C_REAL Class

CLASS	C_REAL	
Purpose	Constraint on instances of `Real`.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**list**: `Set<Real>`	Set of Reals specifying constraint
0..1 (cond)	**range**: `Interval<Real>`	Range of Real specifying constraint
Functions	**Signature**	**Meaning**
(redefined)	**prototype_value**: `Real`	A generated prototype value from this constraint object.
Invariant	***Consistency***: list /= Void ***xor*** range /= Void	

6.2.6 C_DATE Class

CLASS	C_DATE
Purpose	ISO 8601-compatible constraint on instances of `Date` in the form either of a set of validity values, or an actual date range. 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).

CLASS	C_DATE	
Use	Date ranges are probably only useful for historical dates.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**range**: `Interval <ISO8601_DATE>`	Interval of Dates specifying constraint
0..1 (cond)	**pattern**: `String`	ISO8601-based ADL pattern like "yyyy-??-xx"
Functions	**Signature**	**Meaning**
(redefined)	**prototype_value**: `ISO8601_DATE`	A generated prototype value from this constraint object.
	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.
	validity_is_range: `Boolean` ***ensure*** Result = (range /= Void)	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.
Invariant	***Basic_validity***: range /= Void **xor** pattern /= Void ***Pattern_validity***: pattern /= Void **implies** valid_iso8601_date_constraint_pattern(pattern)	

6.2.7 C_TIME Class

CLASS	C_TIME	
Purpose	ISO 8601-compatible constraint on instances of `Time`. 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).	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**range**: `Interval <ISO8601_TIME>`	Interval of Times specifying constraint

CLASS	C_TIME	
0..1 (cond)	**pattern**: `String`	ISO8601-based ADL pattern like "hh:??:xx"
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.
	validity_is_range: `Boolean` ***ensure*** Result = (range /= Void)	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.
Invariant	***Basic_validity***: range /= Void **xor** pattern /= Void ***Pattern_validity***: pattern /= Void **implies** valid_iso8601_time_constraint_pattern(pattern)	

6.2.8 C_DATE_TIME Class

CLASS	C_DATE_TIME	
Purpose	ISO 8601-compatible constraint on instances of `Date_Time`. There is no validity flag for 'year', since it must always be by definition mandatory in order to have a sensible date/time at all. Syntax expressions of instances of this class include "YYYY-MM-DDT??:??:??" (date/time with optional time) and "YYYY-MM-DDTHH:MM:xx" (date/time, seconds not allowed).	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**range**: `Interval <ISO8601_DATE_TIME>`	Range of Date_times specifying constraint
0..1 (cond)	**pattern**: `String`	ISO8601-based pattern like "yyyy-mm-ddT??:??:??"
Functions	**Signature**	**Meaning**

CLASS	C_DATE_TIME	
(redefined)	**prototype_value**: `ISO8601_DATE_TIME`	A generated prototype value from this constraint object.
	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.
	validity_is_range: `Boolean` ***ensure*** Result = (range /= Void)	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.
Invariant	***Basic_validity***: range /= Void **xor** pattern /= Void ***Pattern_validity***: pattern /= Void **implies** valid_iso8601_date_time_constraint_pattern(pattern)	

6.2.9 C_DURATION Class

CLASS	C_DURATION	
Purpose	ISO 8601-compatible constraint on instances of `Duration`. 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 *open*EHR-wide exception.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
0..1	**range**: `Interval <ISO8601_DURATION>`	Constraint expressed as a range of durations.
0..1	**pattern**: `String`	ISO8601-based pattern. Allowed patterns: P[Y|y][M|m][D|d][T[H|h][M|m][S|s]] or P[W|w]
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.
	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.
	validity_is_range: `Boolean` ***ensure*** Result = (range /= Void)	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.

CLASS	C_DURATION
Invariant	***Basic_validity***: pattern /= Void **or** range /= Void ***Pattern_valid***: pattern /= Void **implies** valid_iso8601_duration_constraint_pattern (pattern)

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 ontology, described in section 8 on page 69. The assertion package is illustrated below in FIGURE 12.

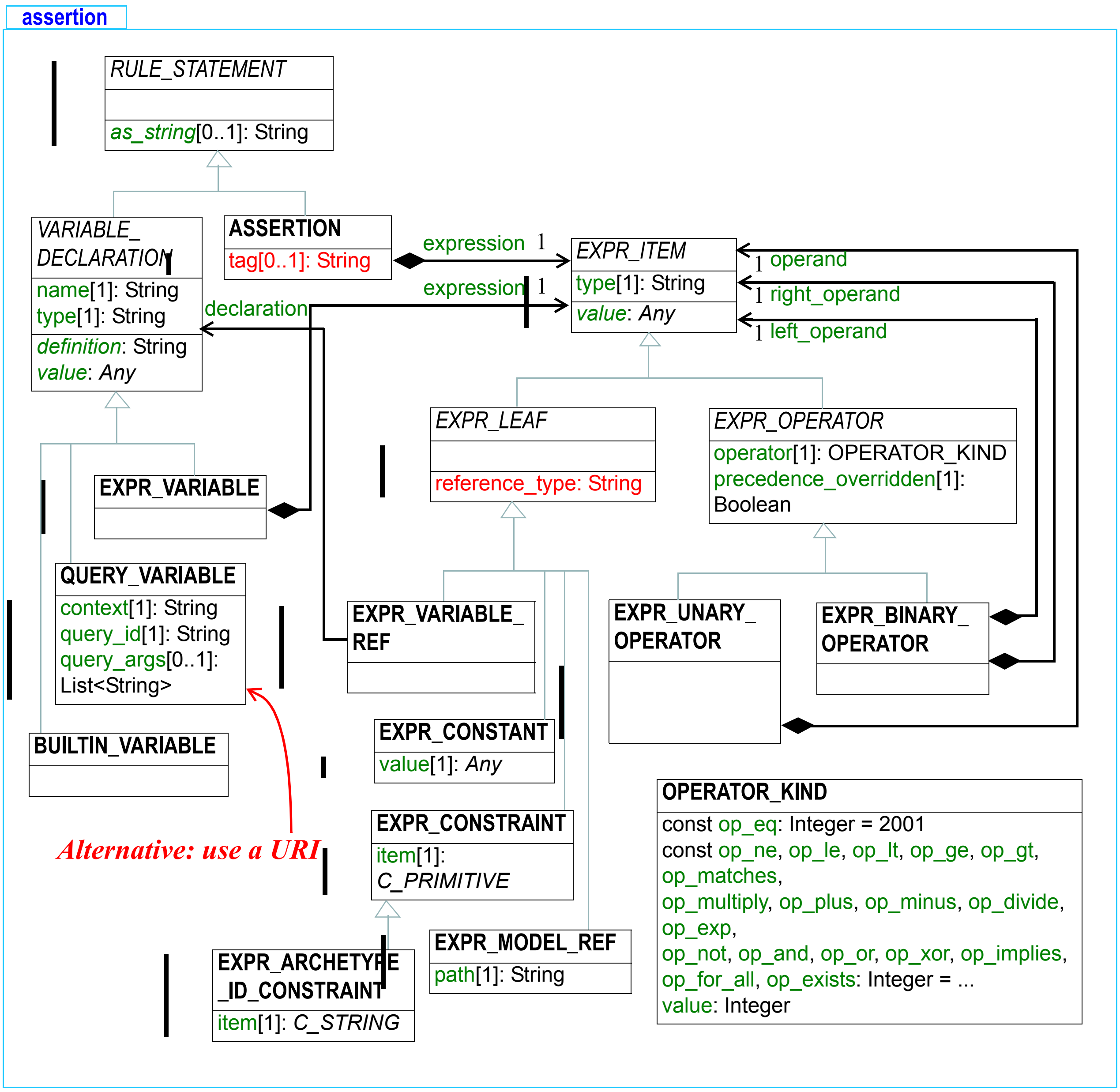


FIGURE 12 The openehr.am.archetype.assertion package

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 *open*EHR 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	***Tag_valid***: tag /= Void **implies not** tag.is_empty ***Expression_valid***: 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
	definition: `String`	Formal definition of the variable.
	value: `Any`	Value of the variable once evaluated.
Attributes	Signature	Meaning
1	**name**: `String`	Name of variable.
1	**type**: `String`	Type of variable, from the *open*EHR assumed types or the *open*EHR reference model.
Invariant	***Name_valid***: name /= Void **and then not** name.is_empty ***Type_valid***: 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	***Expression_valid***: expression /= Void	

7.3.5 BUILTIN_VARIABLE Class

CLASS	BUILTIN_VARIABLE	
Purpose	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: • current_date: `ISO8601_DATE` • current_time: `ISO8601_TIME` • current_date_time: `ISO8601_DATE_TIME`	
Inherit	`VARIABLE_DECLARATION`	
Attributes	Signature	Meaning
Invariant		

7.3.6 QUERY_VARIABLE Class

CLASS	QUERY_VARIABLE	
Purpose	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.	
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	***Context_valid***: context /= Void **implies not** context.is_empty ***Query_id_valid***: query_id /= Void **and then not** query_id.is_empty	

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	***Type_valid***: type /= Void **and then not** type.is_empty	

7.3.8 EXPR_ITEM Class

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

7.3.9 EXPR_CONSTANT Class

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

7.3.10 EXPR_CONSTRAINT Class

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

7.3.11 EXPR_ARCHETYPE_ID_CONSTRAINT Class

CLASS	EXPR_ARCHETYPE_ID_CONSTRAINT
Purpose	Expression tree leaf item representing a constraint on an archetype identifier.

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	***Constraint_validity***: item.is_pattern -- **and** item.pattern matches ARCHETYPE_ID.pattern_template	

7.3.12 EXPR_MODEL_REF Class

CLASS	EXPR_MODEL_REF	
Purpose	Expression tree leaf item representing a reference to a value found in data at a location specified by a path in the archetype definition. • A path referring to a value in the archetype (paths with a leading '/' are in the definition section. • Paths with no leading '/' are in the outer part of the archetype, e.g. "archetype_id/value" refers to the String value of the *archetype_id* attribute of the enclosing archetype.	
Inherit	EXPR_ITEM	
Attributes	Signature	Meaning
1	**path**: String	The path.
Invariant	***Path_valid***: 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	***Declaration_valid***: 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	***operand_valid***: 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	***left_operand_valid***: operand /= Void ***right_operand_valid***: 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 ‘/=’ 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` ***ensure*** an_op >= op_eq **and** an_op <= op_exp	Function to test operator values.
Invariant	***Validity***: valid_operator(value)	

8 Ontology Package

8.1 Overview

All linguistic and terminological entities in an archetype are represented in the ontology part of an archetype, whose semantics are given in the Ontology package, shown below.

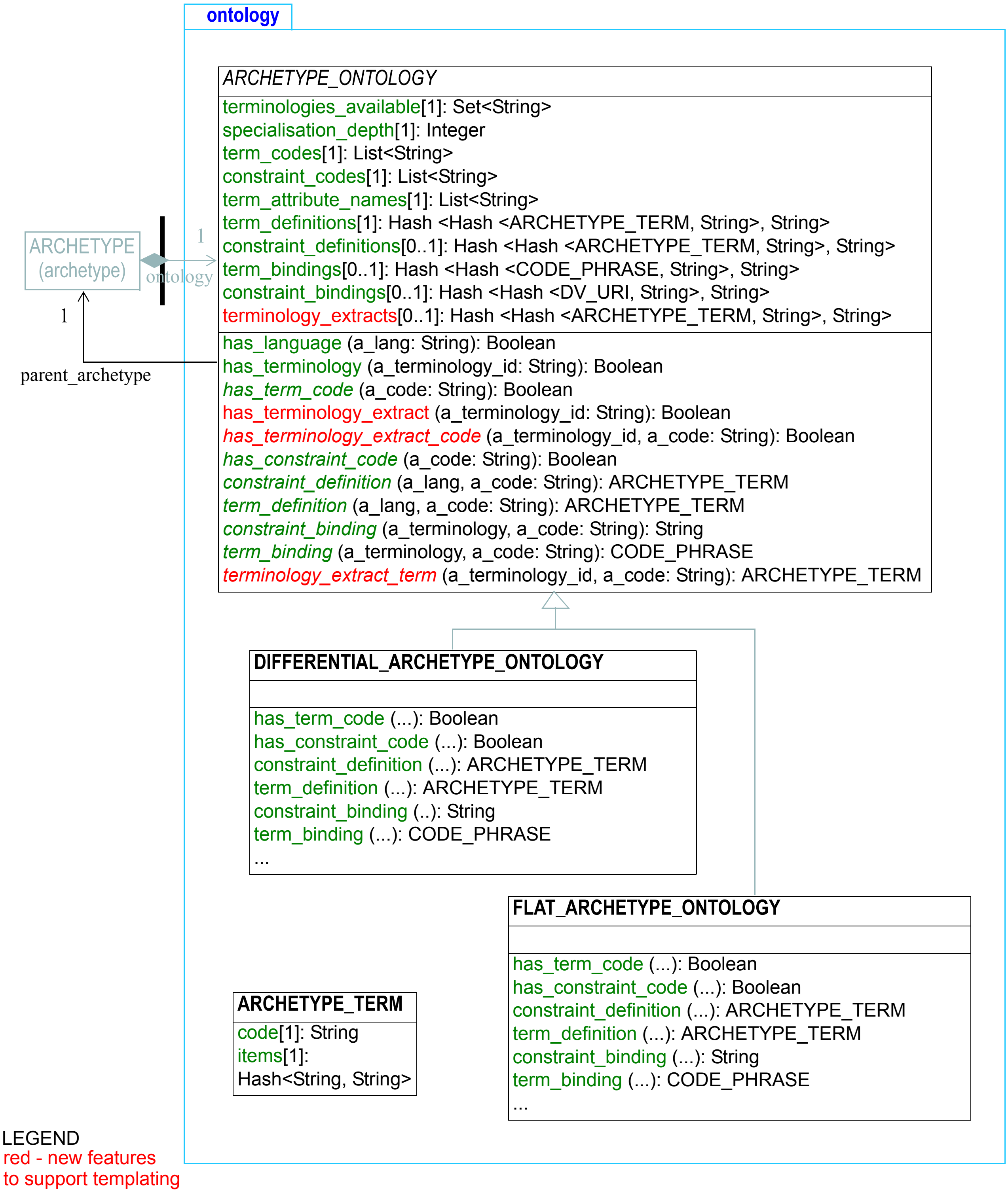


FIGURE 13 openehr.am.archetype.ontology Package

An archetype ontology consists of the following elements.

- A list of terms defined local to the archetype. These are identified by 'atNNNN' codes, and perform the function of archetype node identifiers from which paths are created. There is one such list for each natural language in the archetype. A term 'at0001' defined in English as 'blood group' is an example.
- A list of external constraint definitions, identified by 'acNNNN' codes, for constraints defined external to the archetype, and referenced using an instance of a `CONSTRAINT_REF`. There is one such list for each natural language in the archetype. A term 'ac0001' corresponding to 'any term which is-a blood group', which can be evaluated against some external terminology service.
- Optionally, a set of one or more bindings of term definitions to term codes from external terminologies.
- Optionally, a set of one or more bindings of the external constraint definitions to external resources such as terminlogies.
- Optionally, 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. Such extracts are nearly always added due to localised templating, and correspond to small value sets for which no external reference set or subset is defined.

The differential variant of the `ARCHETYPE_ONTOLOGY` class defines an archetype ontology that only contains terms, constraints, bindings and terminology extracts that were introduced in the owning archetype, whereas the flat variant contains all codes and bindings obtained by compressing an archetype lineage through inheritance. The structure of both forms of `ARCHETYPE_ONTOLOGY` is illustrated in FIGURE 14.

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 indicated by the *specialisation_depth* attribute. An archetype which is not a specialisation of another has a specialisation_depth of 0. Term and constraint codes *introduced* in the ontology of specialised archetypes (i.e. which did not exist in the ontology 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:

- 'at0.0.1' - a new term introduced in this archetype, which is not a specialisation of any previous term in any of the parent archetypes;
- 'at0001.0.1' - a term which specialises the 'at0001' term from the top parent. An intervening '.0' is required to show that the new term is at depth 2, not depth 1;
- 'at0001.1.1' - a term which specialises the term 'at0001.1' from the immediate parent, which itself specialises the term 'at0001' 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.

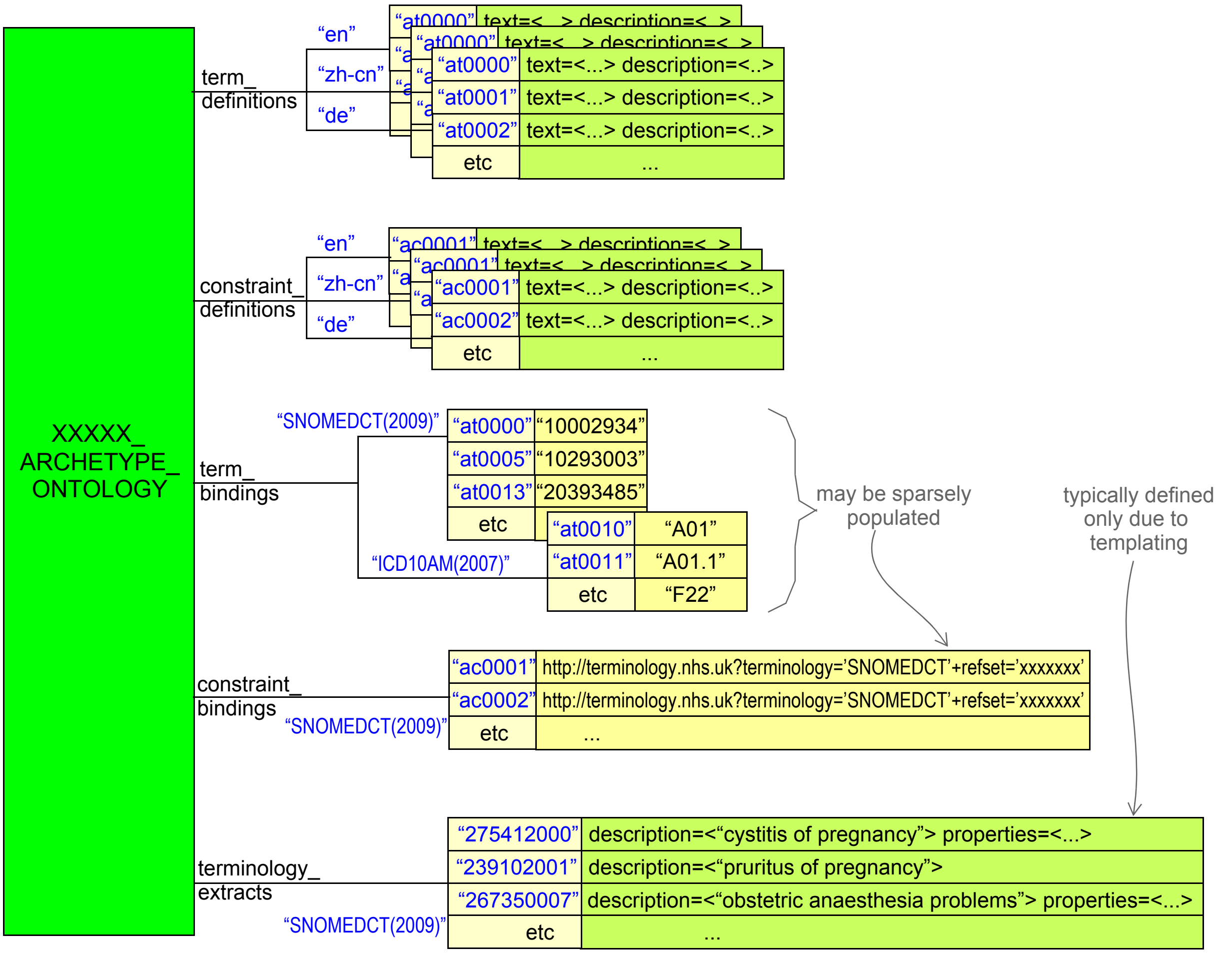


FIGURE 14 Archetype ontology structure.

8.2.2 Term and Constraint Definitions

Local term and constraint definitions are modelled as instances of the class ARCHETYPE_TERM, which is a code associated with a list of name/value pairs. For any term or constraint definition, this list must at least include the name/value pairs for the names “text” and “description”. It might also include such things as “provenance”, which would be used to indicate that a term was sourced from an external terminology. The attribute *term_attribute_names* in ARCHETYPE_ONTOLOGY provides a list of attribute names used in term and constraint definitions in the archetype, including “text” and “description”, as well as any others which are used in various places.

8.3 Class Descriptions

8.3.1 ARCHETYPE_ONTOLOGY Class

CLASS	ARCHETYPE_ONTOLOGY (abstract)	
Purpose	Local ontology of an archetype. This abstract class defines nearly all the semantics of the ontology of an archetype. It is specialised into differential and flat subtypes which implement some routines and supply various different validation semantics.	
Attributes	**Signature**	**Meaning**
1	**terminologies_available**: `Set<String>`	List of terminologies to which term or constraint bindings exist in this terminology.
1	**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.
1	**term_codes**: `List<String>`	List of all term codes in the ontology. Most of these correspond to “at” codes in an ADL archetype, which are the *node_ids* on `C_OBJECT` descendants. There may be an extra one, if a different term is used as the overall archetype *concept* from that used as the node_id of the outermost `C_OBJECT` in the definition part.
1	**constraint_codes**: `List<String>`	List of all term codes in the ontology. These correspond to the “ac” codes in an ADL archetype, or equivalently, the `CONSTRAINT_REF`.*reference* values in the archetype definition.
1	**term_attribute_names**: `List<String>`	List of ‘attribute’ names in ontology terms, typically includes ‘text’, ‘description’, ‘provenance’ etc.
1	**parent_archetype**: `ARCHETYPE`	Archetype which owns this ontology.
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. `“at0004”`.
0..1	**constraint_definitions**: `Hash <Hash <ARCHETYPE_TERM, String>, String>`	Directory of constraint 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. `“at0004”`.

CLASS	ARCHETYPE_ONTOLOGY (abstract)	
0..1	**term_bindings**: Hash <Hash <CODE_PHRASE, String>, String>	Directory of term bindings as a two-level table. The outer hash keys are terminology ids, e.g. "SNOMED_CT", and the inner hash keys are term codes, e.g. "at0004" etc. The indexed CODE_PHRASE objects represent the bound external codes, e.g. Snomed or ICD codes in string form, e.g. "SNOMED_CT::10094842".
0..1	**constraint_bindings**: Hash <Hash <DV_URI, String>, String>	Directory of constraint bindings 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. "ac0004" etc. The indexed DV_URI objects represent references to externally defined resources, usually a terminology subset.
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".
Abstract	**Signature**	**Meaning**
	has_term_code (a_code: String): Boolean	True if *term_codes* has *a_code*.
	has_constraint_code (a_code: String): Boolean	True if *constraint_codes* has *a_code*.
	has_terminology_extract_code (a_terminology_id, a_code: String): Boolean ***require*** has_terminology_extract (a_terminology_id)	True if *terminology_extracts* has for *a_terminology* has *a_code*.
	term_definition (a_lang, a_code: String): ARCHETYPE_TERM ***require*** has_language (a_lang) has_term_code (a_code)	Term definition for a code, in a specified language.

CLASS	ARCHETYPE_ONTOLOGY (abstract)	
	constraint_definition (a_lang, a_code: `String`): `ARCHETYPE_TERM` ***require*** has_language(a_lang) has_constraint_code(a_code)	Constraint definition for a code, in a specified language.
	term_binding (a_terminology_id, a_code: `String`): `CODE_PHRASE` ***require*** has_terminology (a_terminology_id) **and** has_term_code (a_code)	Binding of term corresponding to *a_code* in target external terminology *a_terminology_id* as a `CODE_PHRASE`.
	constraint_binding (a_terminology_id, a_code: `String`): `String` ***require*** has_terminology (a_terminology_id) **and** has_constraint_code (a_code)	Binding of constraint corresponding to *a_code* in target external terminology *a_terminology_id*, as a string, which is usually a formal query expression.
	terminology_extract_term (a_terminology_id, a_code: `String`): `ARCHETYPE_TERM` ***require*** 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.
Functions	**Signature**	**Meaning**
	has_language (a_lang: `String`): `Boolean` ***require*** a_lang /= Void	True if language 'a_lang' is present in archetype ontology.
	has_terminology (a_terminology_id: `String`): `Boolean` ***require*** a_terminology_id /= Void	True if terminology *a_terminology* is present in archetype ontology.

CLASS	ARCHETYPE_ONTOLOGY (abstract)	
	has_terminology_extract (a_terminology_id: `String`): `Boolean` ***require*** a_terminology_id /= Void	True if there is a terminology extract for *a_terminology* is present in archetype ontology.
Invariant	***terminologies_available_exists***: terminologies_available /= void ***term_codes_validity***: term_codes /= void ***constraint_codes_validity*** constraint_codes /= void ***term_definitions_validity***: term_definitions /= void ***constraint_definitions_validity*** constraint_definitions /= void **implies not** constraint_definitions.is_empty ***term_bindings_validity***: term_bindings /= void **implies not** term_bindings.is_empty ***constraint_bindings_validity***: constraint_bindings /= void **implies not** constraint_bindings.is_empty ***term_attribute_names_valid***: term_attribute_names /= void **and then** term_attribute_names.has(“text”) **and** term_attribute_names.has(“description”) ***Parent_archetype_valid***: parent_archetype /= Void and then parent_archetype.description = Current	

8.3.1.1 Validity Rules

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

> **VONSD: specialisation level of codes**. Term or constraint code defined in archetype ontology must be of the same specialisation level as the archetype (differential archetypes), or the same or a less specialised level (flat archetypes).
>
> **VONLC: language consistency**. Languages consistent: all term codes and constraint codes exist in all languages.

8.3.2 DIFFERENTIAL_ARCHETYPE_ONTOLOGY Class

CLASS	DIFFERENTIAL_ARCHETYPE_ONTOLOGY	
Purpose	Differential form of an archetype ontology, containing only codes and bindings introduced in the current archetype.	
Functions	**Signature**	**Meaning**
(effected)	**has_term_code** (a_code: `String`): `Boolean`	
(effected)	**has_constraint_code** (a_code: `String`): `Boolean`	
(effected)	**term_definition** (a_lang, a_code: `String`): `ARCHETYPE_TERM`	

CLASS	DIFFERENTIAL_ARCHETYPE_ONTOLOGY	
(effected)	**constraint_definition**(a_lang, a_code: String): ARCHETYPE_TERM	
(effected)	**term_binding** (a_terminology_id, a_code: String): CODE_PHRASE	
(effected)	**constraint_binding** (a_terminology_id, a_code: String): String	
Invariant		

8.3.3 FLAT_ARCHETYPE_ONTOLOGY Class

CLASS	FLAT_ARCHETYPE_ONTOLOGY	
Purpose	Flat form of an archetype ontology, containing codes and bindings from all archetypes in the inheritance lineage of the current archetype.	
Functions	Signature	Meaning
(effected)	**has_term_code** (a_code: String): Boolean	
(effected)	**has_constraint_code** (a_code: String): Boolean	
(effected)	**term_definition** (a_lang, a_code: String): ARCHETYPE_TERM	
(effected)	**constraint_definition** (a_lang, a_code: String): ARCHETYPE_TERM	
(effected)	**term_binding** (a_terminology_id, a_code: String): CODE_PHRASE	
(effected)	**constraint_binding** (a_terminology_id, a_code: String): String	
Invariant		

8.3.4 ARCHETYPE_TERM Class

CLASS	ARCHETYPE_TERM	
Purpose	Representation of any coded entity (term or constraint) in the archetype ontology.	
Attributes	**Signature**	**Meaning**
1	**code**: `String`	Code of this term.
1	**items**: `Hash <String, String>`	Hash of keys (“`text`”, “`description`” etc) and corresponding values.
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 ***items_valid***: items /= Void	

Appendix A Domain-specific Extension Example

A.1 Overview

Domain-specific classes can be added to the archetype constraint model by inheriting from the class `C_DOMAIN_TYPE`. This section provides an example of how domain-specific constraint classes are added to the archetype model. Actual additions to the AOM for *open*EHR are documented in the *open*EHR Archetype Profile (oAP) specification.

A.2 Scientific/Clinical Computing Types

FIGURE 15 shows the general approach, used to add constraint classes for commonly used concepts in scientific and clinical computing, such as 'ordinal' (used heavily in medicine, particularly in pathology testing), 'coded term' (also heavily used in clinical computing) and 'quantity', a general scientific meansurement concept. The constraint types shown are `C_ORDINAL`, `C_CODED_TEXT` and `C_QUANTITY` which can optionally be used in archetypes to replace the default constraint semantics represented by the use of instances of `C_OBJECT` / `C_ATTRIBUTE` to constrain ordinals, coded terms and quantities. The following model is intended only as an example, and does not try to define any normative semantics of the particular constraint types shown.

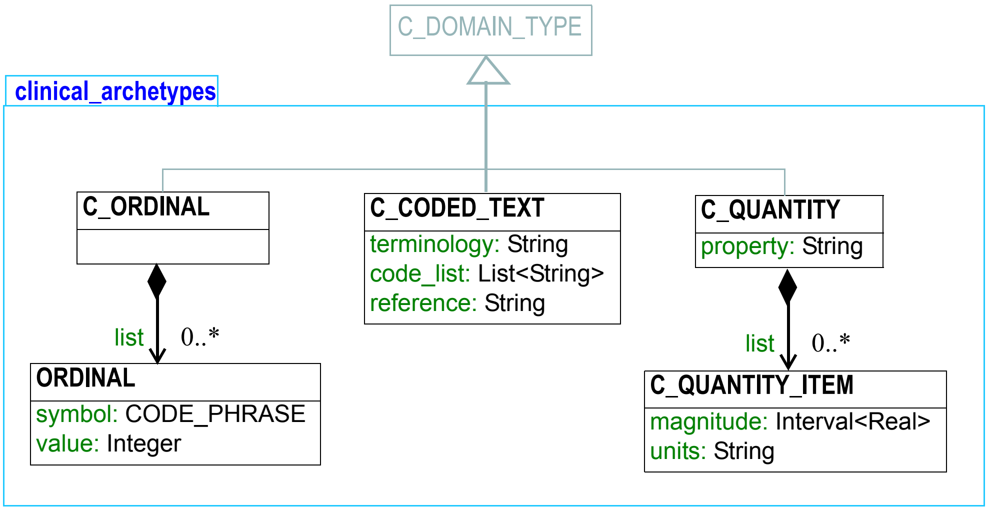


FIGURE 15 Example Domain-specific Package

Appendix B Algorithms

B.1 Validation of Specialised Archetype

The following class provides an indicative algorithm that can be used to validate a specialised archetype against the flat form of its specialisation parent. It is expressed in a Pascal-style notation dervived from the Eiffel reference implementation of the ADL compiler developed for the *open*EHR Foundation. The code and keywords should be self-explanatory, except possibly in the case of the 'agent' keyword. This is used in Eiffel to pass a routine as an object to another routine. The C# equivalent is the 'delegate'; in Java there are various workarounds. The original code can be found at THIS URL.

The design approach of the following class is quite simple: traverse the tree structure of the differential form of a specialised archetype with an agent (delegate) that finds the equivalent node in the flat parent, and determines whether the child node conforms or not.

```
class ARCHETYPE_VALIDATOR

    target: DIFFERENTIAL_ARCHETYPE
          -- differential archetype being validated

    flat_parent: FLAT_ARCHETYPE
          -- flat version of parent archetype, if target is specialised

    validate_specialised_definition is
          -- validate definition of specialised archetype against flat parent
       require
          Target_specialised: target.is_specialised
       local
          def_it: C_ITERATOR
       do
          create def_it.make(target.definition)
          def_it.do_while(agent specialised_node_validate, agent node_test)
       end

    node_test (a_c_node: ARCHETYPE_CONSTRAINT): BOOLEAN  is
          -- return True if a conformant path of a_c_node within the differential archetype is
          -- found within the flat parent archetype - i.e. a_c_node is inherited or redefined from
          -- parent (but not new) and no previous errors encountered
       local
          apa: ARCHETYPE_PATH_ANALYSER
       do
          create apa.make_from_string(a_c_node.path)
          Result := passed and flat_parent.has_path (apa.path_at_level (flat_parent.specialisation_depth))
       end

    specialised_node_validate (a_c_node: ARCHETYPE_CONSTRAINT; depth: INTEGER)
          -- perform grafts of node from differential archetype on corresponding node in flat parent
          -- only interested in C_COMPLEX_OBJECTs
       local
```

```
		co_parent_flat, co_child_diff: C_OBJECT
		apa: ARCHETYPE_PATH_ANALYSER
		child_attr_name: STRING
		ca_parent, ca_child, ca_child_diff: C_ATTRIBUTE
	do
		create apa.make_from_string (a_c_node.path)

		if a_c_node instance_of C_ATTRIBUTE then
			ca_child_diff := (C_ATTRIBUTE) a_c_node
			ca_parent_flat ?= flat_parent.definition.c_attribute_at_path (apa.path_at_level (flat_parent.specialisation_depth))
			if not ca_child_diff.node_conforms_to(ca_parent_flat) then
				if ca_child_diff.is_single /= ca_parent_flat.is_single then
					add_error("VSAM", <<ca_child_diff.path>>)
				elseif not ca_child_diff.existence_conforms_to (ca_parent_flat) then
					add_error("VSANCE", <<ca_child_diff.path, ca_child_diff.existence.as_string,
							ca_parent_flat.path, ca_parent_flat.existence.as_string>>)
				elseif not ca_child_diff.cardinality_conforms_to (ca_parent_flat) then
					add_error("VSANCC", <<ca_child_diff.path, ca_child_diff.cardinality.as_string,
							ca_parent_flat.path, ca_parent_flat.cardinality.as_string>>)
				end
			elseif ca_child_diff.node_congruent_to (ca_parent_flat) and ca_child_diff.parent.is_congruent then
				ca_child_diff.set_is_congruent
			end

		elseif a_c_node instance_of C_OBJECT then
			co_child_diff := (C_OBJECT) a_c_node

			-- find corresponding node in parent by using child node path, 'de-specialised' by one level
			co_parent_flat := flat_parent.c_object_at_path (apa.path_at_level (flat_parent.specialisation_depth))

			-- C_CODE_PHRASE conforms to CONSTRAINT_REF, but is not testable in any way;
			-- sole exception in ADL/AOM; just warn
			if co_parent_flat instance_of CONSTRAINT_REF and not co_child_diff instance_of CONSTRAINT_REF then
				if co_child_diff instance_of C_CODE_PHRASE then
					add_warning("WCRC", <<co_child_diff.path>>)
				else
					add_error("VSCNR", <<co_parent_flat.generating_type, co_parent_flat.path, co_child_diff.generating_type,
										co_child_diff.path>>)
				end
			else
				-- if the child is a redefine of a parent use_node, then have to do the comparison to the
				-- use_node target, unless they both are use_nodes, in which case leave them as is
				if co_parent_flat instance_of ARCHETYPE_INTERNAL_REF and
							not co_child_diff instance_of ARCHETYPE_INTERNAL_REF then
					co_parent_flat := flat_parent.c_object_at_path ((ARCHETYPE_INTERNAL_REF) co_parent_flat.path)
					if dynamic_type (co_child_diff) /= dynamic_type (co_parent_flat) then
						add_error("VSUNT", <<co_child_diff.path, co_child_diff.generating_type,
								co_parent_flat.path, co_parent_flat.generating_type>>)
					end
```

```
                end

                -- now determine if child object is same as or a specialisation of flat object
                if dynamic_type (co_child_diff) /= dynamic_type (co_parent_flat) then
                   add_error("VSONT", <<co_child_diff.path, co_child_diff.type, co_parent_flat.path, co_parent_flat.type>>)
                elseif not co_child_diff.node_conforms_to(co_parent_flat) then
                   if not co_child_diff.rm_type_conforms_to (co_parent_flat) then
                      add_error("VSONCT", <<co_child_diff.path, co_child_diff.rm_type_name,
                                                          co_parent_flat.path, co_parent_flat.rm_type_name>>)
                   elseif not co_child_diff.occurrences_conforms_to (co_parent_flat) then
                      add_error("VSONCO", <<co_child_diff.path, co_child_diff.occurrences.as_string,
                                                          co_parent_flat.path, co_parent_flat.occurrences.as_string>>)
                   elseif co_child_diff.is_addressable then
                      if not co_child_diff.node_id_conforms_to (co_parent_flat) then
                         add_error("VSONCI", <<co_child_diff.path, co_child_diff.node_id, co_parent_flat.path,
                                                                   co_parent_flat.node_id>>))
                      elseif co_child_diff.node_id.is_equal(co_parent_flat.node_id) then
                         add_error("VSONIR", <<co_child_diff.path, co_parent_flat.path, co_child_diff.node_id>>))
                      end
                   else
                      add_error("VSONI", <<co_child_diff.rm_type_name, co_child_diff.path,
                                                          co_parent_flat.rm_type_name, co_parent_flat.path>>))
                   end
                else
                   -- nodes are at least conformant; check for congruence for specalisation path replacement
                   if co_child_diff instance_of C_COMPLEX_OBJECT then
                      if co_child_diff.node_congruent_to (co_parent_flat) and
                                                    (co_child_diff.is_root or else co_child_diff.parent.is_congruent) then
                         co_child_diff.set_is_congruent
                      end
                   end

                   if co_child_diff.sibling_order /= Void and then not
                      co_parent_flat.parent.has_child_with_id (co_child_diff.sibling_order.sibling_node_id) then
                      add_error("VSSM", <<co_child_diff.path, co_child_diff.sibling_order.sibling_node_id>>))
                   end
                end
            end
        end
    end
end
```

B.2 Inheritance-flattening

8.3.5 What is a Redefined Node?

8.3.5.1 Correspondence of Redefined Nodes

Formally speaking, the correspondence of redefined nodes to the parent archetype nodes from which they are derived can be determined according to the following rules.

1. For an identified node in the parent archetype (i.e. at least any child of a container attribute and multiple same-typed children of single-valued attributes), the specialised archetype includes one or more nodes carrying a specialised node identifier, at a congruent path position.
2. For a non-identified node in the parent archetype (i.e. an unidentified child node of a single-valued attribute), the following conditions apply.
   a) Where the node in the parent is the only child node of the attribute, the specialised archetype can include one or more nodes at the corresponding location, whose types *conform* (in the sense of the reference model) to that of the parent node,

To Be Determined: provided that for any type for which there is more than one such node, each node of that type carries a specialised node identifier. [Extension node code maybe?]

   b) Where more than one such child node exists in the parent (each of which must be of different reference model types, by the identification rules described in the ADL specification Summary of Object Node Identification Rules), a specialised node in the child is matched to the parent node of the same or most immediate parent type from the reference model.
   c) Where there are multiple nodes in the parent under the single-valued attribute, and multiple nodes in the child at the same location, matching of specialised nodes to parent nodes may become ambiguous, if reference model subtypes are used.

The above rules are used to determine the lineage of a given node in a specialised archetype, which is required both for archetype validation and for archetype flattening. In case 2c, archetype authoring tools should indicate ambiguities to the authoring user, and potentially offer to add node identifiers in order to remove the ambiguity. For most archetypes and reference models, the use of non-identified nodes is likely to be limited, and such ambiguities will not arise. However for models and archetypes where single-valued attribute alternatives are heavily used and redefined, it is advisable that node identifiers be used both in the parent and specialised child archetypes.

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 *open*EHR. 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