openEHR Release 1.0 public comment

The Archetype Object Model

Editor:T Beale[1]

Contributors: {A Goodchild, Z Tun}[2]*, {T Austin, D Kalra, N Lea, D Lloyd}*[3]

Revision: 2.0

Pages: 53

Keywords: EHR, health records, constraints, archetypes

1. Ocean Informatics Australia (OceanInformatics.biz)
2. Distributed Systems Technology Centre, Brisbane, Australia (DSTC.edu.au)
3. Centre for Health Informatics and Multi-disciplinary Education (CHIME), UCL, London (CHIME.ucl.ac.uk)

The *open*EHR foundation

is an independent, non-profit community, facilitating the creation and 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 1.0			
2.0	**CR-000153**. Synchronise ADL and AOM attribute naming. **CR-000178**. Add Template Object Model to AM. Text changes only. **CR-000167**. Add AUTHORED_RESOURCE class. Remove description package to resource package in Common IM.	T Beale 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

1 Introduction

1.1 Purpose

This document describes a generic object model for archetypes, based only upon the generally accepted semantics of object models (typified by the OMG UML meta-model). 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

Related documents include:

- The *open*EHR Archetype Definition Language (ADL)
- The *open*EHR Archetype Profile (oAP)

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/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 Background

1.5.1 What is an Archetype?

Archetypes are constraint-based models of domain entities, or what some might call "structured business rules". Each archetype describes configurations of data instances whose classes are described in a reference model; the instance configurations are considered to be valid exemplars of a particular domain concept. Thus, in medicine, an archetype might be designed to constrain configurations of instances of a simple node/arc information model, that express a "microbiology test result" or a "physical examination". Archetypes can be composed, specialised, and templated for local use. The archetype concept has been described in detail by Beale [1], [2]. Most of the detailed formal semantics are described in the *open*EHR Archetype Definition Language [4]. The *open*EHR archetype framework is described in terms of Archetype Definitions and Principles [4] and an Archetype System [5].

1.5.2 Context

The object model described in this document relates to linguistic forms of archetypes as shown in FIGURE 1. The model (upper right in the figure) is the object-oriented semantic equivalent of the ADL the Archetype Definition Language BNF language definition, and, by extension, any formal transformation of it. Instances of the model (lower right on the figure) are themselves archetypes, and correspond one-to-one with archetype documents expressed in ADL or a related language.

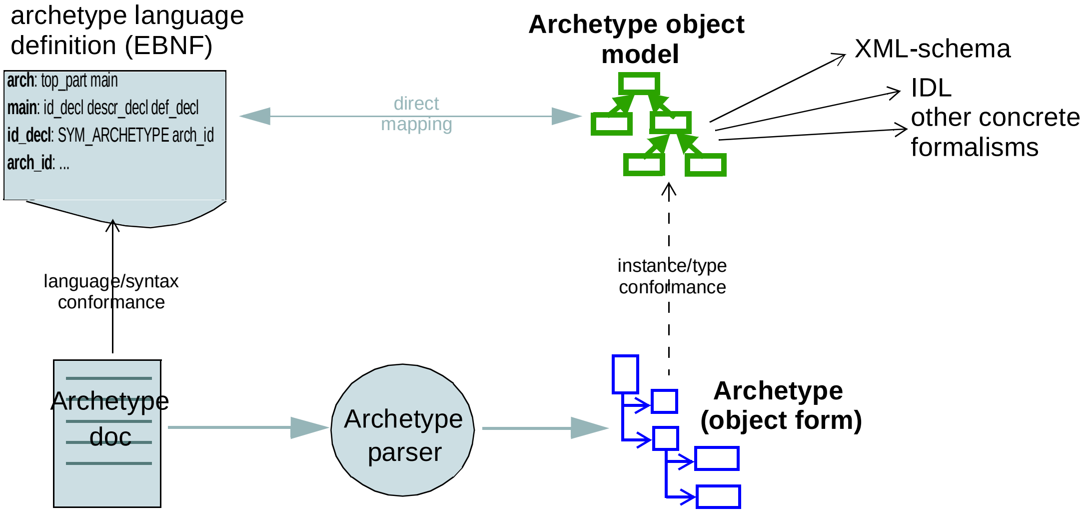

FIGURE 1 Relationship of Archetype Object Model to Archetype Languages

1.6 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`).

1.7 Changes from Previous Versions

1.7.1 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 The Archetype Object Model

2.1 Design Background

An underpinning principle of *open*EHR is the use of archetypes and templates, which are formal models of domain concepts controlling data structure and content of data. The elements of this architecture are twofold.

- The *open*EHR Reference Model (RM), defining the structure and semantics of information, and the service models (SMs), describing service interfaces. These models correspond respectively to the ISP RM/ODP information and computational viewpoints. The information models define the data of *open*EHR EHR systems; meaning that every data instance in a system is an instance of a type defined in the Information Model (or to be completely correct, the corresponding type in the relevant ITS). The information model is designed to be invariant in the long term, to minimise the need for software and schema updates.
- The *open*EHR Archetype Model (AM), defining the structure and semantics of archetypes and templates. The AM consists of the archetype language definition language (ADL), the Archetype Object Model (AOM), the *open*EHR Archetype profile (OAP) and the Template Object Model (TOM).

The purpose of ADL is to provide an abstract syntax for textually expressing archetypes and templates. The AOM defines the object model equivalent, in terms of a UML model. It is a *generic* model, meaning that it can be used to express archetypes for any reference model in a standard way. ADL and the AOM are brought together in an ADL parser: a tool which can read ADL archetype texts, and whose parse-tree (resulting in-memory object representation) is instances of the AOM. The TOM defines the object model of templates, which are themselves used to put archetypes together into local information structures, usually corresponding to screen forms.

The purpose of the *open*EHR Archetype Profile is to define which classes and attributes of the *open*EHR RM can sensibly archetyped, and to provide custom archetype classes.

2.2 Package Structure

The *open*EHR Archetype Object Model is defined as the package `am.archetype`, as illustrated in FIGURE 2. It is shown in the context of the *open*EHR `am.archetype` packages.

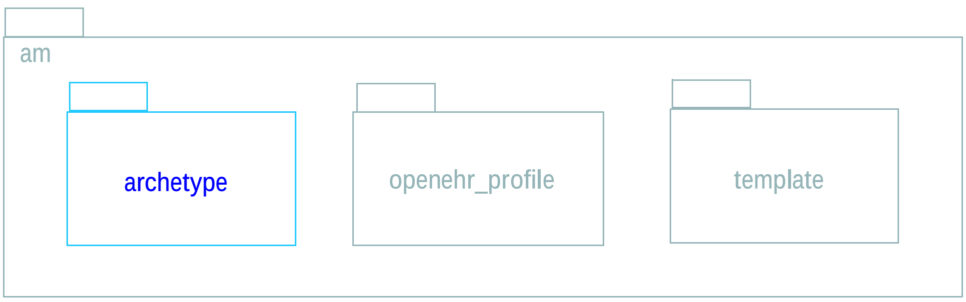

FIGURE 2 openehr.am.archetype Package

2.3 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 only on two groups of assumed types. The first group includes the following primitive inbuilt types, whose names and assumed semantics are described by ISO 11404 (the exact *open*EHR correspondences are described in the *open*EHR Support Information Model).

- `Boolean`
- `Character`
- `Integer`

The second are assumed library types:

- `String`
- `Date`
- `Time`
- `Date_time`
- `Duration`
- `Hash <T, K:Comparable> (keyed list of items of any type)`
- `Interval <T:Comparable> (interval of instances of any ordered type)`

These types are supported in most implementation technologies, including XML, Java and other programming languages. They are not defined in this specification, allowing them to be mapped to the most appropriate concrete types in each implementation technology.

The *open*EHR types used are:

- `ARCHETYPE_ID`
- `HIER_OBJECT_ID`
- `TERMINOLOGY_ID`
- `CODE_PHRASE`
- `DV_CODED_TEXT`

The last of these can be found in the `openEHR RM.Data_types.Text` package, and is used only to represent coded term values in the model. The remaining `*_ID` types can be found in the `openEHR RM.Common.Identification` package.

2.3.1 Archetypes as Objects

FIGURE 3 illustrates various processes that can be responsible for creating an archetype object structure, including parsing, database retrieval and GUI editing. A parsing process that would typically turn a syntax expression of an archetype (ADL, XML, OWL) into an object one. The input file is converted by a parser into an object parse tree, shown on the right of the figure, whose types are specified in this document. Database retrieval will cause the reconstruction of an archetype in memory from a structured data representation, such as relational data, object data or XML. Direct in-memory editing by a user with a GUI archetype editor application will cause on-the-fly creation and destruction of parts of an archetype during the editing session, which would eventually cause the archetype to be stored in some form when the user decides to commit it.

As shown in the figure, the definition part of the in-memory archetype consists of alternate layers of *object* and *attribute* constrainer nodes, each containing the next level of nodes. 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

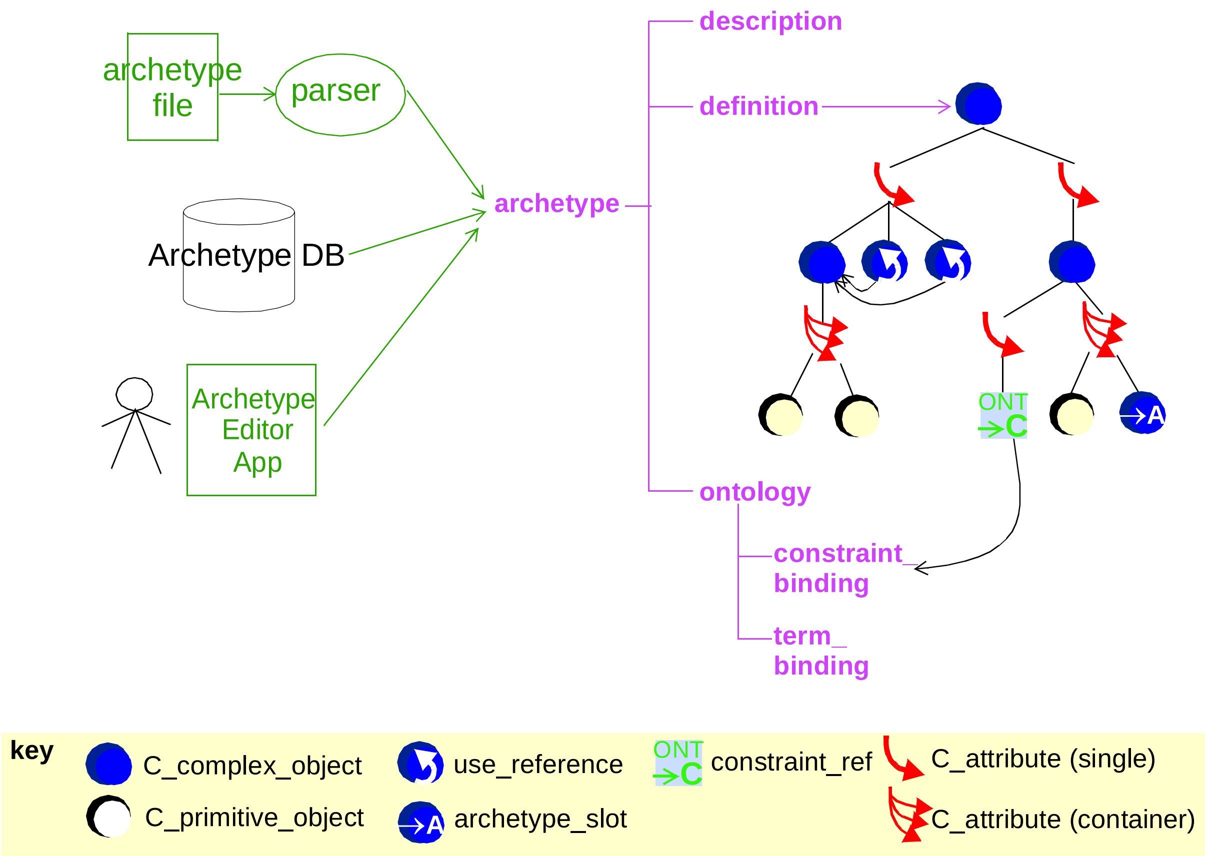

FIGURE 3 Archetype Parsing Process

model. 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 node types is as follows:

C_complex_object: any interior node representing a constraint on instances of some non-primitive type, e.g. `ENTRY`, `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.

The typename nomenclature “C_complex_object”, “C_primitive_object”, “C_attribute” used here is intended to be read as “constraint on xxxx”, i.e. a “C_complex_object” is a “constraint on a complex object (defined by a complex reference model type)”. These typenames are used below in the formal model.

2.3.2 The Archetype Ontology

There are no linguistic entities at all in the definition part of an archetype, with the possible exception of constraints on text items which might have been defined in terms of regular expression patterns or fixed strings. All linguistic entities are defined in the ontology part of the archetype, in such a way as to allow them to be translated into other languages in convenient blocks. As described in the *open*EHR ADL document, there are four major parts in an archetype ontology: term definitions, constraint definitions, term bindings and constraint bindings. The former two define the meanings of various terms and textual constraints which occur in the archetype; they are indexed with unique identifiers which are used within the archetype definition body. The latter two ontology sections describe the mappings of terms used internally to external terminologies. Due to the well-known problems with terminologies (described in some detail in the *open*EHR ADL document, and also by e.g. Rector [6] and others), mappings may be partial, incomplete, approximate, and occasionally, exact.

2.3.3 Archetype Specialisation

Archetypes can be specialised. The formal rules of specialisation are described in the *open*EHR Archetype Semantics document (forthcoming), but in essence are easy to understand. Briefly, an archetype is considered 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. Any data created via the use of the specialised archetype is thus conformant both to it and its parent. This notion of specialisation corresponds to the idea of ‘substitubility’, applied to data.

Every archetype has a ‘specialisation depth’. Archetypes with no specialisation parent have depth 0, and specialised archetypes add one level to their depth for each step down a hierarchy required to reach them.

2.3.4 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 a multitude of others. Archetype slots are the means of composition, and are themselves defined in terms of constraints.

2.4 The Archetype Package

2.4.1 Overview

The model of an archetype, illustrated in FIGURE 4, 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, and an *ontology*. The archetype definition, the 'main' part of an archetype, is an

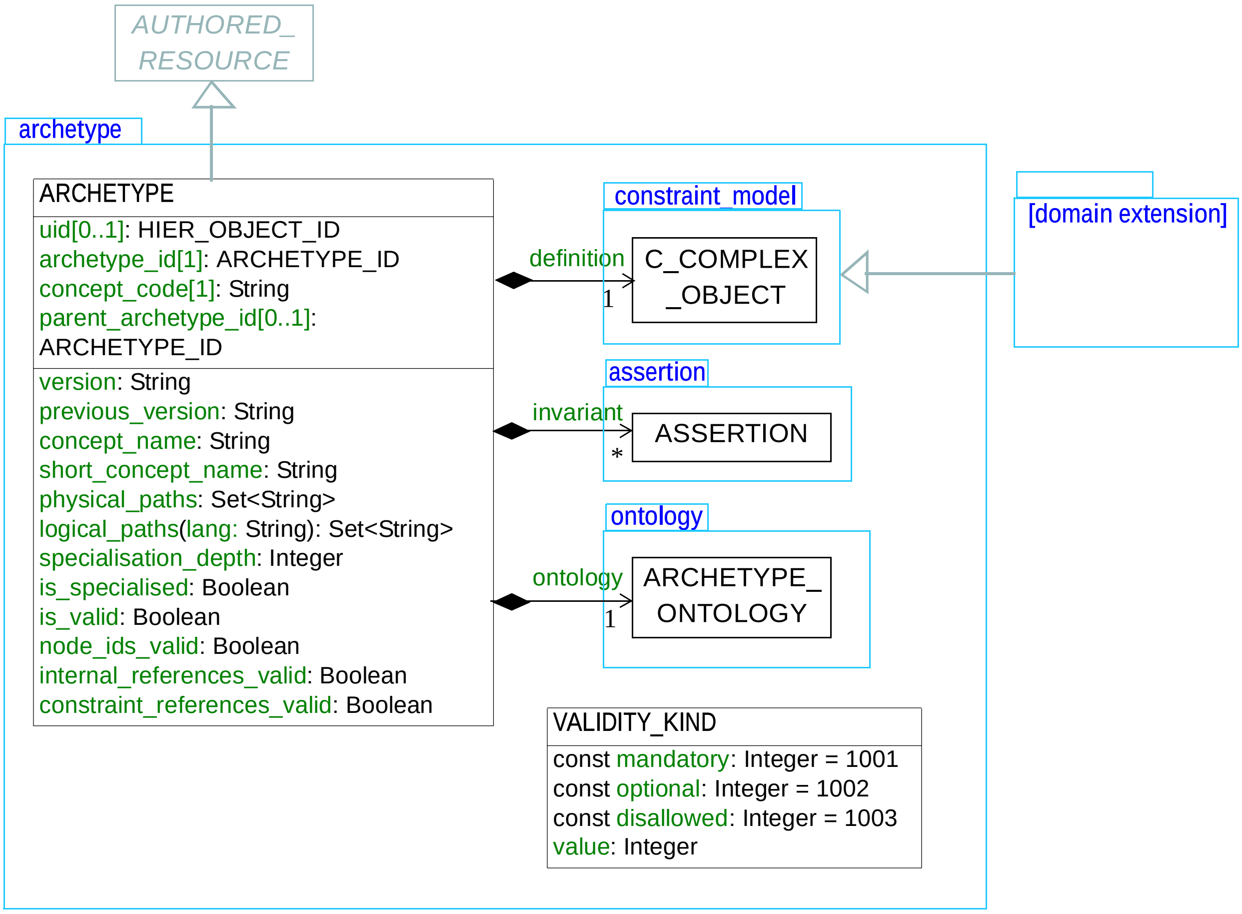

FIGURE 4 openehr.am.archetype Package

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`.

2.4.2 ARCHETYPE Class

CLASS	ARCHETYPE	
Purpose	Archetype equivalent to `ARCHETYPED` class in Common reference model. Defines semantics of identfication, lifecycle, versioning, composition and specialisation.	
Inherit	`AUTHORED_RESOURCE`	
Attributes	**Signature**	**Meaning**
	archetype_id: `ARCHETYPE_ID`	Multi-axial identifier of this archetype in archetype space.
	uid: `HIER_OBJECT_ID`	OID identifier of this archetype.
	concept_code: `String`	The normative meaning of the archetype as a whole, has the same value as the *concept_code* in the archetype ontology; corresponds to the *title* feature.
	parent_archetype_id: `ARCHETYPE_ID`	Identifier of the specialisation parent of this archetype.
	definition: `C_COMPLEX_OBJECT`	Root node of this archetype
	ontology: `ARCHETYPE_ONTOLOGY`	The ontology of the archetype.
Functions	**Signature**	**Meaning**
	version: `String`	Version of this archetype, extracted from id.
	previous_version: `String`	Version of predecessor archetype of this archetype, if any.
	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_code* attribute in the archetype ontology.
	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.

CLASS	ARCHETYPE	
	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.
	is_specialised: `Boolean` ***ensure*** *Result* **implies** parent_archetype_id /= Void	True if this archetype is a specialisation of another.
	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 *ontol-ogy.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.
Invariant	***archetype_id_validity***: archetype_id /= Void ***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("languages").has(original_language) ***description_exists***: description /= Void ***definition_exists***: definition /= Void ***ontology_exists***: ontology /= Void ***Specialisation_validity***: is_specialised **implies** specialisation_depth > 0	

2.4.3 VALIDITY_KIND Class

CLASS	VALIDITY_KIND	
Purpose	An enumeration of three values which may commonly occur in constraint models.	
Use	Use as the type of any attribute within this model, which expresses constraint on some attribute in a class in a reference model. For example to indicate validity of Date/Time fields.	
Attributes	**Signature**	**Meaning**
	***const* mandatory**: `Integer = 1001`	Constant to indicate mandatory presence of something
	***const* optional**: `Integer = 1002`	Constant to indicate optional presence of something
	***const* disallowed**: `Integer = 1003`	Constant to indicate disallowed presence of something
	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)	

2.5 Constraint Model Package

2.5.1 Overview

FIGURE 5 illustrates the class model of 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'.

One way to comprehend the model is via the following statements that can be made about it.

- Any archetype definition is an instance of a `C_COMPLEX_OBJECT`, which can be thought of as expressing constraints on a object that is of some particular type (recorded in the attribute *rm_type_name*) in a reference model, and which is larger than a simple instance of a primitive type such as String or Integer.
- A `C_COMPLEX_OBJECT` 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`.
- The key subtypes of `C_OBJECT`, are `C_COMPLEX_OBJECT` (described above) `C_PRIMITIVE_OBJECT` (constraints on instances of primitive types such as String, Integer, Boolean and Date).
- The other subtypes of `C_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.
- All nodes in an archetype constraint structure are instances of the supertype `ARCHETYPE_CONSTRAINT`, which provides a number of important common features to all nodes.

2.5.2 Semantics

The effect of the model is to create archetype description structures that are a hierarchical alternation of object and attribute constraints, as shown in FIGURE 3. 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.

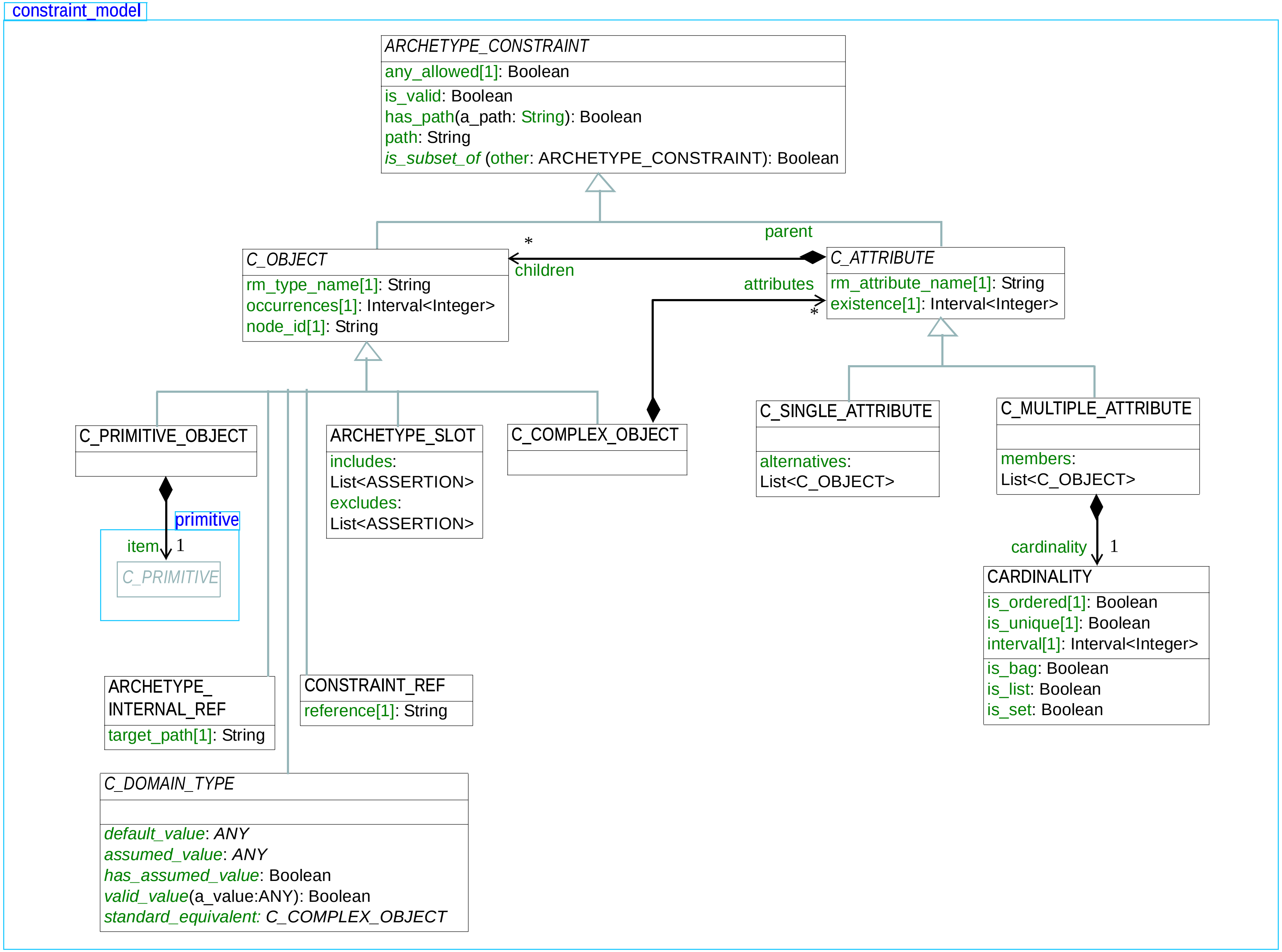

FIGURE 5 openehr.am.archetype.constraint_model Package

All Node Types

A small number of properties is defined for all node types. The *any_allowed* flag set 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. 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 *is_valid* function indicates whether the current node and all subnodes are internally valid according to the semantics of this archetype model.

Attribute Nodes

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 6 illustrates the two possibilities.

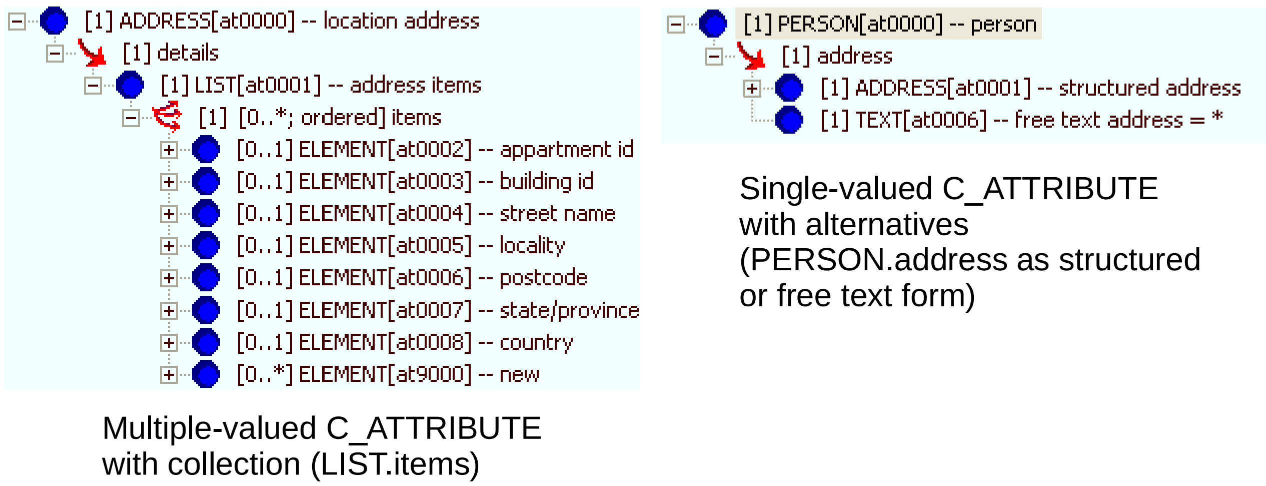

FIGURE 6 Single and Multiple-valued C_ATTRIBUTES

The need for both *existence* and *cardinality* constraints in the `C_MULTIPLE_ATTRIBUTE` class deserves some explanation, especially as the meanings of these notions are often confused in object-oriented literature. Quite simply, 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 required. 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.

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.

Constraint References

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 (such as an ontology or 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

To Be Determined: whether this apporach should be used instead:

To Be Determined: The other problem is that the CONSTRAINT_REF could probably stand for a C_PRIMITIVE_OBJECT, such as a plain C_string or C_integer (which can still be a little bit complex - e.g. a Interval<Integer>).

To Be Determined: Following on logically from this, a more correct modelling prossibility might be to introduce a common parent for C_COMPLEX_OBJECT and C_PRIMITIVE_OBJECT which corresponds to the idea of C_OBJECTs 'defined by value in the archetype' (as opposed to defined elsewhere, like in binding query to a terminology, or else in an entirely differnt archetype, which is what the slot gives you). If I introduced such a type, then CONSTRAINT_REF should have a property called something nlike 'proxy_for' or 'equivalent', which points to this new type, allowing it to stand for either a primitive or complex constraint structure. Now that you have driven me to think of that, I see it as being quite a good improvement - maybe Andrew will have feedback on it.

To Be Determined: Another parent is probably needed for C_DOMAIN_TYPE and C_PRIMITIVE, to coalesce their leaf attributes assumed_value etc. This would probably require merging C_PRIMITIVE_OBJECT and C_PRIMITIVE.

Assertions

The C_ATTRIBUTE and subtypes of C_OBJECT enable constraints to be expressed in a structural fashion. In addition to this, any instance of a C_COMPLEX_OBJECT may include one or more *invariants*. Invariants are statements in a form of predicate logic, which can be used to state constraints on parts of an object. They are not needed to state constraints on a single attribute (since this can be done with an appopriate C_ATTRIBUTE), but are necessary to state constraints on more than one attribute, such as a constraint that ‘systolic pressure should be >= diastolic pressure’ in a blood pressure measurement archetype. Invariants are expressed using a syntax derived from the OMG’s OCL syntax (adapted for use with objects rather than classes).

To Be Continued: give decent example

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.

Node_id and Paths

The *node_id* attribute in the class C_OBJECT, inherited to 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 is what 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*.

Domain-specific Extensions

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 47.

Assumed Values

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.

2.5.3 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.	
Attributes	**Signature**	**Meaning**
	any_allowed: `Boolean`	True if any instance value of this type is considered valid in this archetype. Allows completely 'open' constraints to be expressed without requiring any further structure.
Functions	**Signature**	**Meaning**
	is_valid: `Boolean`	True if this node (and all its sub-nodes) is a valid archetype node for its type. This function should be implemented by each subtype to perform semantic validation of itself, and then call the is_valid function in any sub-parts, and generate the result appropriately.
	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.
	is_subset_of (other: `ARCHETYPE_CONSTRAINT`): `Boolean` ***require*** other /= Void	True if constraints represented by *other* are narrower than this node.
Invariant	***path_exists***: path /= Void	

To Be Continued: Note: is_subset_of is relatively easy to evaluate for structures, but acNNNN constraints and assertions will be harder, and will most likely require evaluation in a subsumptive environment like OWL.

2.5.4 C_ATTRIBUTE Class

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

CLASS	C_ATTRIBUTE(abstract)	
	rm_attribute_name: `String`	Reference model attribute within the enclosing type represented by a `C_OBJECT`.
	existence: `Interval<Integer>`	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).
	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.
Invariant	***Rm_attribute_name_valid***: rm_attribute_name /= Void **and then not** rm_attribute_name.is_empty ***Existence_set***: existence /= Void **and then** (existence.lower >= 0 **and** existence.upper <= 1) ***Children_validity***: any_allowed **xor** children /= Void	

2.5.5 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_exists***: alternatives /= Void	

2.5.6 C_MULTIPLE_ATTRIBUTE Class

CLASS	C_MULTIPLE_ATTRIBUTE	
Purpose	Abstract model of constraint on any kind of attribute node.	
Attributes	**Signature**	**Meaning**
	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*.
Invariant	***Cardinality_validity***: cardinality /= Void ***Members_valid***: members /= Void **and then** members.for_all(co: `C_OBJECT` | co.occurrences.upper <= 1)	

2.5.7 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**
	is_ordered: `Boolean`	True if the members of the container attribute to which this cardinality refers are ordered.
	is_unique: `Boolean`	True if the members of the container attribute to which this cardinality refers are unique.
	interval: `Interval<Integer>`	The interval of this cardinality.
Attributes	**Signature**	**Meaning**
	is_bag: `Boolean` ***ensure*** *Result* = **not** is_ordered **and not** 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_set `Boolean` ***ensure*** *Result* = **not** is_ordered **and** is_unique	True if the semantics of this cardinality represent a bag, i.e. unordered, non-unique membership.
Invariant	***Validity***: **not** interval.lower_unbounded	

2.5.8 C_OBJECT Class

CLASS	*C_OBJECT (abstract)*	
Purpose	Abstract model of constraint on any kind of object node.	
Attributes	Signature	Meaning
	rm_type_name: `String`	Reference model type that this node corresponds to.
	occurrences: `Interval<Integer>`	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).
	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.
	parent: `C_ATTRIBUTE`	`C_ATTRIBUTE` that owns this `C_OBJECT`.
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	

2.5.9 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_OBJECT`	
Attributes	Signature	Meaning
	attributes: `Set<C_ATTRIBUTE>`	List of constraints on attributes of the reference model type represented by this object.
	invariants: `Set<ASSERTION>`	Invariant statements about this object. Statements are expressed in first order predicate logic, and usually refer to at least two attributes.
Invariant	***attributes_valid***: any_allowed **xor** (attributes /= Void **and not** attributes.is_empty) ***invariant_consistency***: any_allowed **implies** invariants = Void ***invariants_valid***: invariants /= Void **implies not** invariants.is_empty	

2.5.10 ARCHETYPE_SLOT Class

CLASS	ARCHETYPE_SLOT	
Purpose	Constraint describing a ‘slot’ where another archetype can occur.	
Inherit	`C_OBJECT`	
Attributes	**Signature**	**Meaning**
	includes: `Set <ASSERTION>`	List of constraints defining other archetypes that could be included at this point.
	excludes: `Set<ASSERTION>`	List of constraints defining other archetypes that cannot be included at this point.
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	

2.5.11 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.	
Inherit	`C_OBJECT`	
Attributes	**Signature**	**Meaning**
	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)	

2.5.12 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_OBJECT`	
Attributes	**Signature**	**Meaning**

CLASS	CONSTRAINT_REF	
	reference: `String`	Reference to a constraint in the archetype local ontology.
Invariant	***Consistency:*** **not** any_allowed ***reference_valid***: reference /= Void **and then not** reference.is_empty **and then** archetype.ontology.has_constraint(reference)	

2.5.13 C_PRIMITIVE_OBJECT Class

CLASS	C_PRIMITIVE_OBJECT	
Purpose	Constraint on a primitive type.	
Inherit	`C_OBJECT`	
Attributes	Signature	Meaning
	item: `C_PRIMITIVE`	Object actually defining the constraint.
Invariant	***item_exists***: any_allowed **xor** item /= Void	

2.5.14 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_OBJECT	
Abstract	Signature	Meaning
	default_value: ANY	Generate a default value from this constraint object
	has_assumed_value: Boolean	True if there is an assumed value
	assumed_value: ***like*** default_value	Value to be assumed if none sent in data
	valid_value (a_value: **like** default_value): Boolean ***require*** a_value /= Void	True if a_value is valid with respect to constraint expressed in concrete instance of this type.
	standard_representation: C_COMPLEX_OBJECT	Standard form of constraint
Invariant	***Assumed_value_valid***: valid_value(assumed_value)	

2.6 The Assertion Package

2.6.1 Overview

Assertions are expressed in archetypes in typed first-order predicate logic (FOL). They are used in two places: to express archetype slot constraints, and to express invariants 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 2.8 on page 43.

2.6.2 Semantics

The concrete syntax of assertion statements in archetypes is designed to be compatible with the OMG Object Constraint Language (OCL) [10]. Archetype assertions are essentially statements which contain the following elements:

- *variables*, which are attribute names, or ADL paths terminating in attribute names (i.e. equivalent of referencing class feature in a programming language);
- *manifest constants* of any primitive type, plus 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**

The written syntax of assertions is defined in the *open*EHR ADL document. The package described here is currently designed to allow the representation of a general-purpose binary expression tree, as would be generated by a parser. This may be replaced in the future by a more specific model, if needed. The assertion package is illustrated below in FIGURE 7.

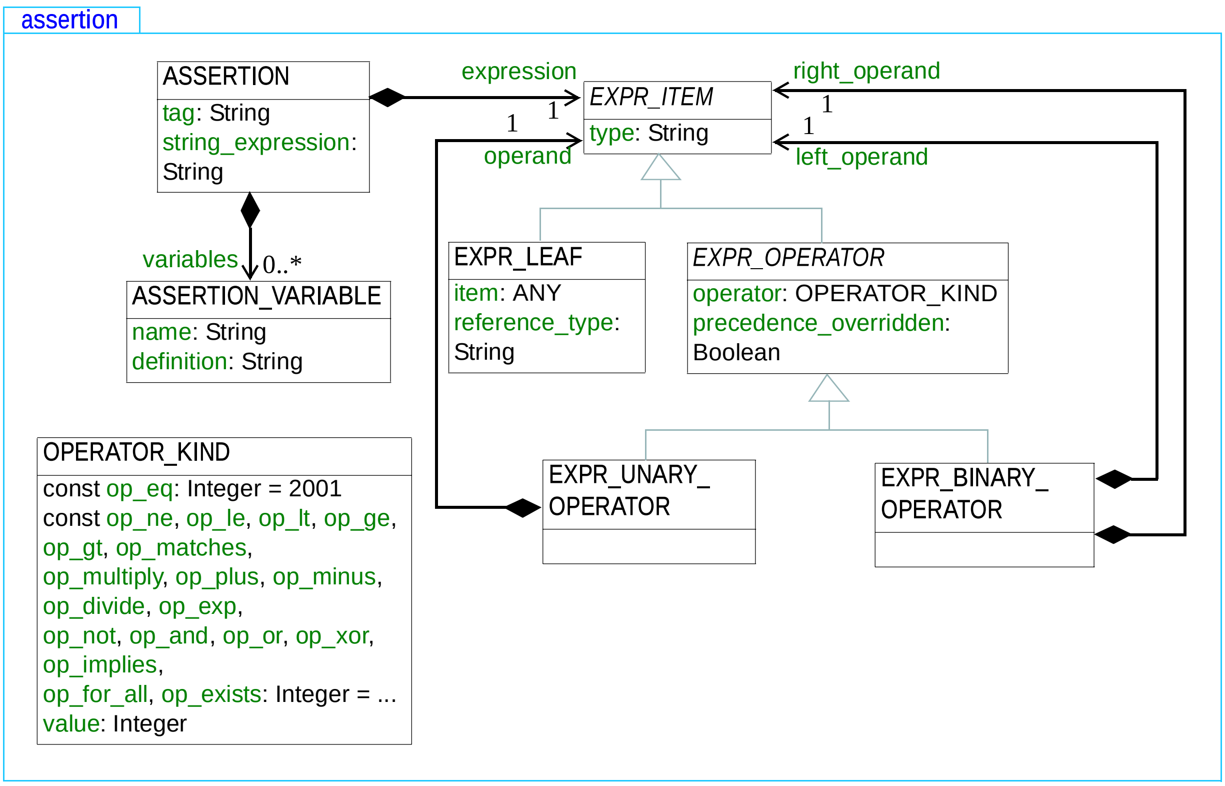

FIGURE 7 The openehr.am.archetype.assertion package

This relatively simple model of expressions is sufficiently powerful for representing FOL expressions on archetype structures, although it could clearly be more heavily subtyped.

2.6.3 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.	
Attributes	**Signature**	**Meaning**
	tag: `String`	Expression tag, used for differentiating multiple assertions.
	expression: `EXPR_ITEM`	Root of expression tree.
	string_expression: `String`	String form of expression, in case an expression evaluator taking String expressions is used for evaluation.
	variables: `List<ASSERTION_VARIABLE>`	Definitions of variables used in the assertion expression.
Invariant	***Tag_valid***: tag /= Void **implies not** tag.is_empty ***Expression_valid***: expression /= Void and then expression.type.is_equal("BOOLEAN")	

2.6.4 EXPR_ITEM Class

CLASS	*EXPR_ITEM (abstract)*	
Purpose	Abstract parent of all expression tree items.	
Attributes	**Signature**	**Meaning**
	type: `String`	Type name of this item. 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	

2.6.5 EXPR_LEAF Class

CLASS	EXPR_LEAF	
Purpose	Expression tree leaf item	
Inherit	EXPR_ITEM	
Attributes	Signature	Meaning
	item: ANY	The value referred to; a manifest constant, an attribute path, or a C_PRIMITIVE. [Future: possibly function names as well, even if not constrained in the archetype - as long as they are in the reference model].
	reference_type: String	Type of reference: “constant”, “attribute”, “function”
Invariant	***Item_valid***: item /= Void	

2.6.6 EXPR_OPERATOR Class

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

2.6.7 EXPR_UNARY_OPERATOR Class

CLASS	EXPR_UNARY_OPERATOR
Purpose	Unary operator expression node.
Inherit	EXPR_OPERATOR

CLASS	EXPR_UNARY_OPERATOR	
Attributes	**Signature**	**Meaning**
	operand: EXPR_ITEM	Operand node.
Invariant	***operand_valid***: operand /= Void	

2.6.8 EXPR_BINARY_OPERATOR Class

CLASS	EXPR_BINARY_OPERATOR	
Purpose	Binary operator expression node.	
Inherit	EXPR_OPERATOR	
Attributes	**Signature**	**Meaning**
	left_operand: EXPR_ITEM	Left operand node.
	right_operand: EXPR_ITEM	Right operand node.
Invariant	***left_operand_valid***: operand /= Void ***right_operand_valid***: operand /= Void	

2.6.9 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)	

2.7 The Primitive Package

Ultimately any archetype definition will devolve down to leaf node constraints on instances of primitive types. The primitive package, illustrated in FIGURE 8, defines the semantics of constraint on such types. 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.

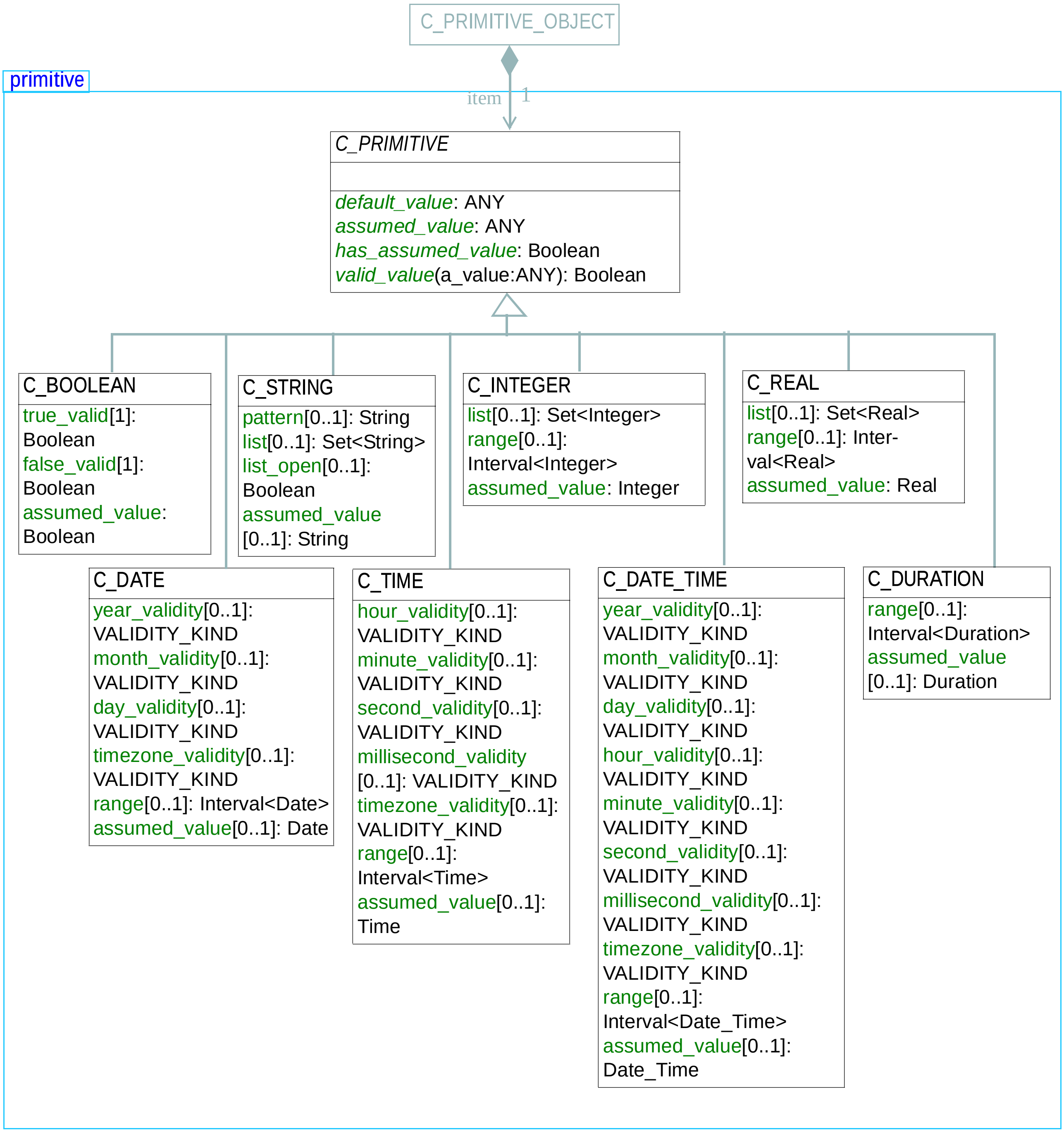

FIGURE 8 The openehr.am.archetype.primitive Package

Action: add State primitive type

2.7.1 C_PRIMITIVE Class

CLASS	*C_PRIMITIVE (abstract)*	
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.	
Abstract	**Signature**	**Meaning**
	default_value: `ANY`	Generate a default value from this constraint object
	has_assumed_value: `Boolean`	True if there is an assumed value
	assumed_value: ***like*** default_value	Value to be assumed if none sent in data
	valid_value (a_value: **like** default_value): `Boolean` ***require*** `a_value /= Void`	True if a_value is valid with respect to constraint expressed in concrete instance of this type.
	standard_representation: `C_COMPLEX_OBJECT`	Standard form of constraint
Invariant	***Assumed_value_valid***: valid_value(assumed_value)	

2.7.2 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**
	true_valid: `Boolean`	True if the value True is allowed
	false_valid: `Boolean`	True if the value False is allowed
	assumed_value: `Boolean`	The value to assume if this item is not included in data, due to being part of an optional structure.

CLASS	C_BOOLEAN
Invariant	***Binary_consistency***: true_valid ***or*** false_valid ***Default_value_consistency***: default_value.value ***and*** true_valid ***or else not*** default_value.value ***and*** false_valid

2.7.3 C_STRING Class

CLASS	C_STRING	
Purpose	Constraint on instances of `STRING`.	
Inherit	`C_PRIMITIVE`	
Attributes	Signature	Meaning
	pattern: `String`	Regular expression pattern for proposed instances of `String` to match.
	list: `Set<String>`	Set of Strings specifying constraint
	list_open: `Boolean`	True if the list is being used to specify the constraint but is not considered exhaustive.
	assumed_value: `String`	The value to assume if this item is not included in data, due to being part of an optional structure.
Invariant	***Consistency***: pattern /= Void **xor** list /= Void ***pattern_exists***: pattern /= Void **implies not** pattern.is_empty	

To Be Continued: TB: is list_open really useful? If the list is open, then what's the difference from 'any_allowed'

2.7.4 C_INTEGER Class

CLASS	C_INTEGER	
Purpose	Constraint on instances of `Integer`.	
Inherit	`C_PRIMITIVE`	
Attributes	Signature	Meaning
	list: `Set<Integer>`	Set of Integers specifying constraint
	range: `Interval<Integer>`	Range of Integers specifying constraint
	assumed_value: `Integer`	The value to assume if this item is not included in data, due to being part of an optional structure.

CLASS	C_INTEGER
Invariant	***Consistency***: list /= Void ***xor*** range /= Void

2.7.5 C_REAL Class

CLASS	C_REAL	
Purpose	Constraint on instances of `Real`.	
Inherit	`C_PRIMITIVE`	
Attributes	Signature	Meaning
	list: `Set<Real>`	Set of Reals specifying constraint
	range: `Interval<Real>`	Range of Real specifying constraint
	assumed_value: `Real`	The value to assume if this item is not included in data, due to being part of an optional structure.
Invariant	***Consistency***: list /= Void ***xor*** range /= Void	

2.7.6 C_DATE Class

CLASS	C_DATE	
Purpose	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.	
Use	Date ranges are probably only useful for historical dates.	
Inherit	`C_PRIMITIVE`	
Attributes	Signature	Meaning
	month_validity: `VALIDITY_KIND`	Validity of month in constrained date.
	day_validity: `VALIDITY_KIND`	Validity of day in constrained date.
	timezone_validity: `VALIDITY_KIND`	Validity of timezone in constrained date.
	range: `Interval<Date>`	Interval of Dates specifying constraint
	assumed_value: `Date`	The value to assume if this item is not included in data, due to being part of an optional structure.

CLASS	C_DATE	
Functions	**Signature**	**Meaning**
	validity_is_range: `Boolean`	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.
Invariant	***Month_validity_optional***: month_validity = `{VALIDITY_KIND}`.optional **implies** (day_validity = `{VALIDITY_KIND}`.optional **or** day_validity = `{VALIDITY_KIND}`.disallowed) ***Month_validity_disallowed***: month_validity = `{VALIDITY_KIND}`.disallowed **implies** day_validity = `{VALIDITY_KIND}`.disallowed ***Validity_is_range***: validity_is_range = (range /= Void)	

2.7.7 C_TIME Class

CLASS	C_TIME	
Purpose	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.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
	minute_validity: `VALIDITY_KIND`	Validity of minute in constrained time.
	second_validity: `VALIDITY_KIND`	Validity of second in constrained time.
	millisecond_validity: `VALIDITY_KIND`	Validity of millisecond in constrained time.
	timezone_validity: `VALIDITY_KIND`	Validity of timezone in constrained date.
	range: `Interval<Time>`	Interval of Times specifying constraint
	assumed_value: `Time`	The value to assume if this item is not included in data, due to being part of an optional structure.
Functions	**Signature**	**Meaning**
	validity_is_range: `Boolean`	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.

CLASS	C_TIME
Invariant	***Minute_validity_optional***: minute_validity = {VALIDITY_KIND}.optional **implies** (second_validity = {VALIDITY_KIND}.optional **or** second_validity = {VALIDITY_KIND}.disallowed) ***Minute_validity_disallowed***: minute_validity = {VALIDITY_KIND}.disallowed **implies** second_validity = {VALIDITY_KIND}.disallowed ***Second_validity_optional***: second_validity = {VALIDITY_KIND}.optional **implies** (millisecond_validity = {VALIDITY_KIND}.optional **or** millisecond_validity = {VALIDITY_KIND}.disallowed) ***Second_validity_disallowed***: second_validity = {VALIDITY_KIND}.disallowed **implies** millisecond_validity = {VALIDITY_KIND}.disallowed ***Validity_is_range***: validity_is_range = (range /= Void)

2.7.8 C_DATE_TIME Class

CLASS	C_DATE_TIME	
Purpose	Constraint on instances of Date_Time. re is no validity flag for 'year', since it must always be by definition mandatory in order to have a sensible date/time at all.	
Inherit	C_PRIMITIVE	
Attributes	**Signature**	**Meaning**
	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.
	range: Interval<Date_Time>	Range of Date_times specifying constraint
	assumed_value: Date_Time	The value to assume if this item is not included in data, due to being part of an optional structure.

CLASS	C_DATE_TIME	
Functions	**Signature**	**Meaning**
	validity_is_range: `Boolean`	True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.
Invariant	***Month_validity_optional***: month_validity = `{VALIDITY_KIND}`.optional **implies** (day_validity = `{VALIDITY_KIND}`.optional **or** day_validity = `{VALIDITY_KIND}`.disallowed) ***Month_validity_disallowed***: month_validity = `{VALIDITY_KIND}`.disallowed **implies** day_validity = `{VALIDITY_KIND}`.disallowed ***Day_validity_optional***: day_validity = `{VALIDITY_KIND}`.optional **implies** (hour_validity = `{VALIDITY_KIND}`.optional **or** hour_validity = `{VALIDITY_KIND}`.disallowed) ***Day_validity_disallowed***: day_validity = `{VALIDITY_KIND}`.disallowed **implies** hour_validity = `{VALIDITY_KIND}`.disallowed ***Hour_validity_optional***: hour_validity = `{VALIDITY_KIND}`.optional **implies** (minute_validity = `{VALIDITY_KIND}`.optional **or** minute_validity = `{VALIDITY_KIND}`.disallowed) ***Hour_validity_disallowed***: hour_validity = `{VALIDITY_KIND}`.disallowed **implies** minute_validity = `{VALIDITY_KIND}`.disallowed ***Minute_validity_optional***: minute_validity = `{VALIDITY_KIND}`.optional **implies** (second_validity = `{VALIDITY_KIND}`.optional **or** second_validity = `{VALIDITY_KIND}`.disallowed) ***Minute_validity_disallowed***: minute_validity = `{VALIDITY_KIND}`.disallowed **implies** second_validity = `{VALIDITY_KIND}`.disallowed ***Second_validity_optional***: second_validity = `{VALIDITY_KIND}`.optional **implies** (millisecond_validity = `{VALIDITY_KIND}`.optional **or** millisecond_validity = `{VALIDITY_KIND}`.disallowed) ***Second_validity_disallowed***: second_validity = `{VALIDITY_KIND}`.disallowed **implies** millisecond_validity = `{VALIDITY_KIND}`.disallowed ***Validity_is_range***: validity_is_range = (range /= Void)	

2.7.9 C_DURATION Class

CLASS	C_DURATION	
Purpose	Constraint on instances of `Duration`.	
Inherit	`C_PRIMITIVE`	
Attributes	**Signature**	**Meaning**
	range: `Interval<Duration>`	Range of Durations specifying constraint
	assumed_value: `Duration`	The value to assume if this item is not included in data, due to being part of an optional structure.
Invariant	***Range_valid***: range /= Void	

2.8 Ontology Package

2.8.1 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.

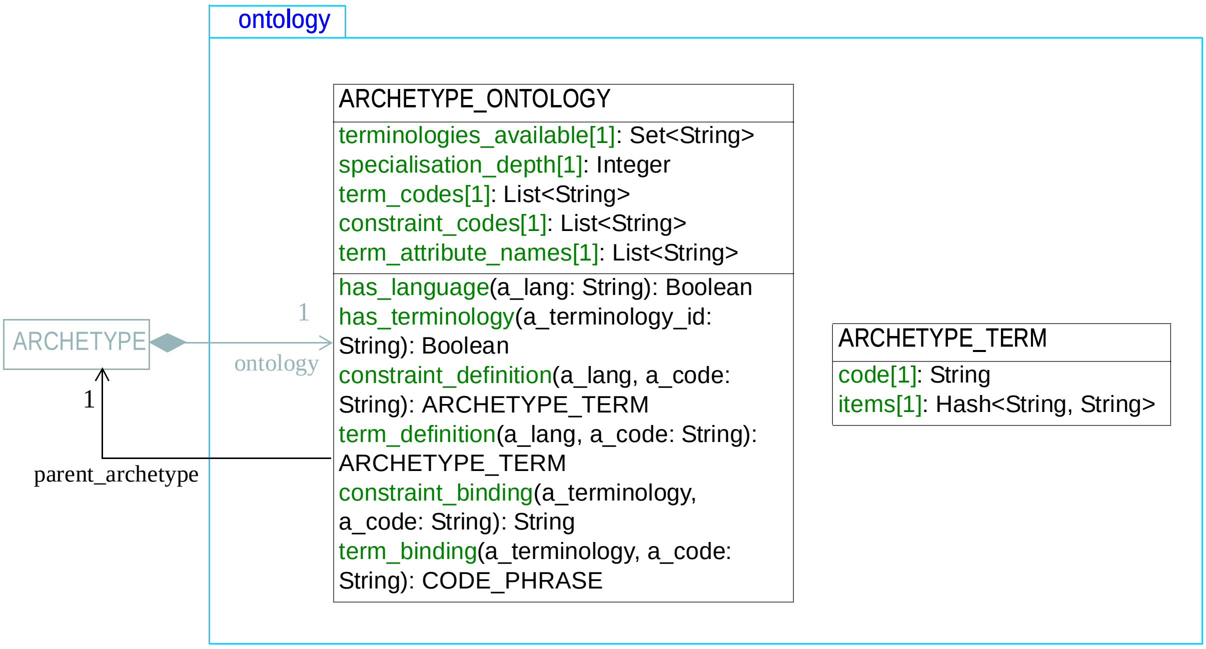

FIGURE 9 openehr.am.archetype.ontology Package

An archetype ontology consists of the following things.

- 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.

2.8.2 Semantics

Specialisation Depth

Any given archetype occurs at some point in a hierarchy 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.

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.

2.8.3 ARCHETYPE_ONTOLOGY Class

CLASS	ARCHETYPE_ONTOLOGY	
Purpose	Local ontology of an archetype.	
Attributes	**Signature**	**Meaning**
	terminologies_available: `Set<String>`	List of terminologies to which term or constraint bindings exist in this terminology.
	specialisation_depth: `Integer`	Specialisation depth of this archetype. Unspecialised archetypes have depth 0, with each additional level of specialisation adding 1 to the specialisation_depth.
	term_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_code from that used as the node_id of the outermost `C_OBJECT` in the definition part.

CLASS	ARCHETYPE_ONTOLOGY	
	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.
	term_attribute_names: List<String>	List of 'attribute' names in ontology terms, typically includes 'text', 'description', 'provenance' etc.
	parent_archetype: ARCHETYPE	Archetype which owns this ontology.
Functions	**Signature**	**Meaning**
	has_language(a_lang: String): Boolean	True if language 'a_lang' is present in archetype ontology.
	has_terminology(a_terminology_id: String): Boolean ***require*** has_terminology(a_terminology_id)	True if terminology 'a_terminology' is present in archetype ontology.
	term_definition(a_lang, a_code: String): ARCHETYPE_TERM ***require*** has_language(a_lang) term_codes.has(a_code)	Term definition for a code, in a specified language.
	constraint_definition(a_lang, a_code: String): ARCHETYPE_TERM ***require*** has_language(a_lang) constraint_codes.has(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) term_codes.has(a_code)	Binding of term corresponding to *a_code* in target external terminology *a_terminology_id* as a CODE_PHRASE.

CLASS	ARCHETYPE_ONTOLOGY	
	constraint_binding (a_terminology_id, a_code: `String`): `String` ***require*** has_terminology(a_terminology_id) constraint_codes.has(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.
Invariant	***terminologies_available_exists***: terminologies_available /= void ***term_codes_exists***: term_codes /= void ***constraint_codes_exists***: constraint_codes /= void ***term_bindings_exists***: term_bindings /= void ***constraint_bindings_exists***: constraint_bindings /= void ***term_attribute_names_valid***: term_attribute_names /= void **and then** term_attribute_names.has("text") **and** term_attribute_names.has("description") ***concept_code_valid***: term_codes.has (concept_code) ***Parent_archetype_valid***: parent_archetype /= Void and then parent_archetype.description = Current	

2.8.4 ARCHETYPE_TERM Class

CLASS	ARCHETYPE_TERM	
Purpose	Representation of any coded entity (term or constraint) in the archetype ontology.	
Attributes	**Signature**	**Meaning**
	code: `String`	Code of this term.
	items: `Hash <String, String>`	Hash of keys ("text", "description" etc) and corresponding values.
Functions	**Signature**	**Meaning**
	keys: `Set<String>`	List of all keys used in this term.
Invariant	***code_valid***: code /= void **and then not** code.is_empty	

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.

A.2 Scientific/Clinical Computing Types

FIGURE 10 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.

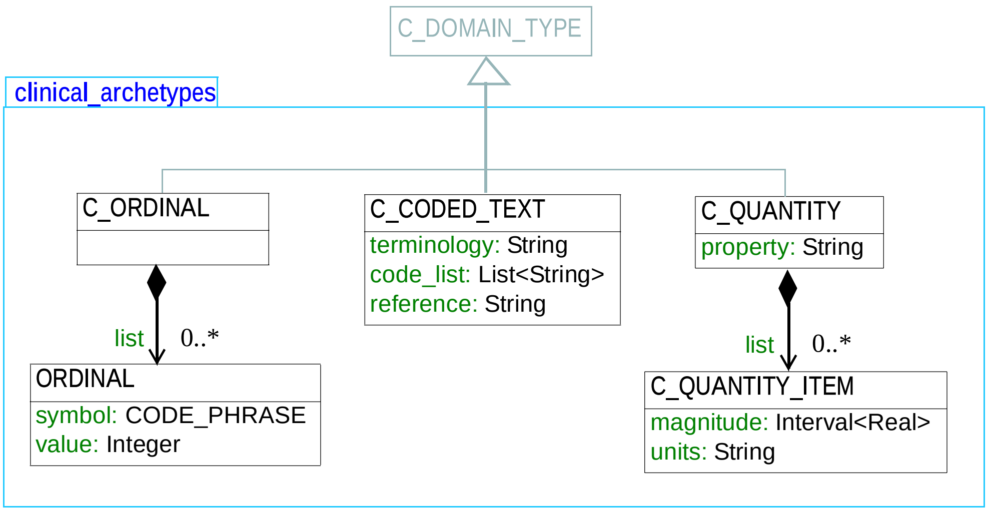

FIGURE 10 Example Domain-specific Package

B Using Archetypes with Diverse Reference Models

B.1 Overview

The archetype model described in this document can be used with any reference model which is expressed in UML or a similar object-oriented formalism. It can also be used with E/R models. The following section describes is use a number of reference models used in clinical computing.

B.2 Clinical Computing Use

To Be Continued:

- data types
- class naming
- domain archetype semantics versus LCD semantics of exchange models
- mapping from `C_DOMAIN_TYPE` subtypes into various RMs

B.2.1 *open*EHR

B.2.2 CEN ENV13606

B.2.3 HL7 Clinical Document Architecture (CDA)

B.2.4 HL7v3 RIM

C 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