REFERENCE MODEL

The *open*EHR Common Information Model

Editors:{T Beale, S Heard}[1], {D Kalra, D Lloyd}[2]

Revision: 1.5

Pages: 45

1. Ocean Informatics Australia

2. Centre for Health Informatics and Multi-professional Education, University College London

Copyright Notice

Amendment Record

Issue	Details	Who	Completed
1.5	CR-000080. Remove ARCHETYPED.*concept* - not needed in data CR-000081. LINK should be unidirectional. CR-000083. RELATED_PARTY.*party* should be optional. CR-000085. LOCATABLE.*synthesised* not needed. Add vocabulary for FEEDER_AUDIT.*change_type*. CR-000086. LOCATABLE.*presentation* not needed. CR-000091. Correct anomalies in use of CODE_PHRASE and DV_CODED_TEXT. Changed PARTICIPATION.*mode*, changed ATTESTATION.*status*, RELATED_PARTY.*relationship*, VERSION_AUDIT.*change_type*, FEEDER_AUDIT.*change_type* to to DV_CODED_TEXT. CR-000094. Add lifecycle state attribute to VERSION; correct DV_STATE. **Formally validated using ISE Eiffel 5.4.**	DSTC T Beale, S Heard DSTC	09 Mar 2004
1.4.12	CR-000071. Allow version ids to be optional in TERMINOLOGY_ID. CR-000044. Add reverse ref from VERSION_REPOSITORY<T> to owner object. CR-000063. ATTESTATION should have a status attribute. CR-000046. Rename COORDINATED_TERM and DV_CODED_TEXT.*definition*.	T Beale D Lloyd D Kalra T Beale	25 Feb 2004
1.4.11	CR-000056. References in COMMON.Version classes should be OBJECT_REFs.	T Beale	02 Nov 2003
1.4.10	CR-000045. Remove VERSION_REPOSITORY.*status*	D Lloyd, T Beale	21 Oct 2003
1.4.9	CR-000025. Allow ATTESTATIONs to attest parts of COMPOSITIONs. Change made due to CEN TC/251 joint WGM, Rome, Feb 2003. CR-000043. Move External package to Common RM and rename to Identification (incorporates CR-000036 - Add HIER_OBJECT_ID class, make OBJECT_ID class abstract.)	D Kalra, D Lloyd, T Beale	09 Oct 2003
1.4.8	CR-000041. Visually differentiate primitive types in openEHR documents.	D Lloyd	04 Oct 2003
1.4.7	CR-000013. Rename key classes according to CEN ENV13606.	S Heard, D Kalra, T Beale	15 Sep 2003
1.4.6	CR-000012. Add *presentation* attribute to LOCATABLE. CR-000027. Move *feeder_audit* to LOCATABLE to be compatible with CEN 13606 revision. Add new class FEEDER_AUDIT.	D Kalra	20 Jun 2003
1.4.5	CR-000020. Move VERSION.*charset* to DV_TEXT, *territory* to TRANSACTION. Remove VERSION.*language*.	A Goodchild	10 Jun 2003
1.4.4	CR-000007. Add RELATED_PARTY class to GENERIC package. CR-000017. Renamed VERSION.*parent_version_id* to *preceding_version_id*.	S Heard, D Kalra	11 Apr 2003

Issue	Details	Who	Completed
1.4.3	**Major alterations** due to CR-000003, CR-000004. ARCHETYPED class no longer inherits from LOCATABLE, now related by association. Redesign of Change Control package. Document structure improved. (Formally validated)	T Beale, Z Tun	18 Mar 2003
1.4.2	Moved External package to Support RM. Corrected CONTRIBUTION.*description* to DV_TEXT. Made PARTICIPATION.*time* optional. (Formally validated).	T Beale	25 Feb 2003
1.4.1	**Formally validated using ISE Eiffel 5.2.** Corrected types of VERSIONABLE.*language*, *charset*, *territory*. Added ARCHETYPED.*uid*:OBJECT_ID. Renamed ARCHETYPE_ID.*rm_source* to *rm_originator*, and *rm_level* to *rm_concept*; added *archetype_originator*. Rewrote archetype id section. Changed PARTICIPATION.*mode* to COORDINATED_TERM & fixed invariant.	T Beale, D Kalra	18 Feb 2003
1.4	**Changes post CEN WG meeting Rome Feb 2003.** Changed ARCHETYPED.*meaning* from STRING to DV_TEXT. Added CONTRIBUTION.*name* invariant. Removed AUTHORED_VA and ACQUIRED_VA audit types, moved feeder audit to the EHR RM. VERSIONABLE.*code_set* renamed to *charset*. Fixed pre/post condition of OBJECT_ID.*context_id*, added OBJECT_ID.*has_context_id*. Changed TERMINOLOGY_ID string syntax.	T Beale, D Kalra, D Lloyd	8 Feb 2003
1.3.5	Removed segment from archetype_id; corrected inconsistencies in diagrams and class texts.	Z Tun, T Beale	3 Jan 2003
1.3.4	Removed inheritance from VERSIONABLE to ARCHETYPED.	T Beale	3 Jan 2003
1.3.3	Minor corrections: OBJECT_ID; changed syntax of TERMINOLOGY_ID. Corrected Fig 6.	T Beale	17 Nov 2002
1.3.2	Added Generic Package; added PARTICIPATION and changed and moved ATTESTATION class.	T Beale	8 Nov 2002
1.3.1	Removed EXTERNAL_ID.*iso_oid*. Remodelled EXTERNAL_ID into new classes - OBJECT_REF and OBJECT_ID. Remodelled all change control classes.	T Beale, D Lloyd, M Darlison, A Goodchild	22 Oct 2002
1.3	Moved ARCHETYPE_ID.*iso_oid* to EXTERNAL_ID. DV_LINK no longer a data type; renamed to LINK.	T Beale	22 Oct 2002
1.2	Removed Structure package to own document. Improved CM diagrams.	T Beale	11 Oct 2002
1.1	Removed HCA_ID. Included Spatial package from EHR RM. Renamed SPATIAL to STRUCTURE.	T Beale	16 Sep 2002
1.0	Taken from EHR RM.	T Beale	26 Aug 2002

Acknowledgements

The work reported in this paper has been funded in by a number of organisations, including The University College, London; The Cooperative Research Centres Program through the Department of the

Prime Minister and Cabinet of the Commonwealth Government of Australia; Ocean Informatics, Australia.

Table of Contents

1 Introduction

1.1 Purpose

This document describes the architecture of the *open*EHR Common Reference Model, which contains concepts used by other *open*EHR reference models.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development groups using *open*EHR;
- Academic groups using *open*EHR;
- The open source healthcare community;
- Medical informaticians and clinicians intersted in health information;
- Health data managers.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Modelling Guide
- The *open*EHR Support Reference Model
- The *open*EHR Data Types Reference Model

1.3 Status

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

Currently the UML diagrams are hand-produced. None of the existing tools (e.g. Rose, Objecteering), includes sufficient support of UML or has good enough visual quality to use here. However, UML tools are constantly under investigation, and this situation may change in the future.

The latest version of this document can be found in PDF and HTML formats at http://www.openEHR.org/Doc_html/Model/Reference/common_rm.htm. New versions are announced on openehr-announce@openehr.org.

1.4 Peer review

Areas where more analysis or explanation is required are indicated with “to be continued” paragraphs like the following:

To Be Continued: more work required

Reviewers are encouraged to comment on and/or advise on these paragraphs as well as the main content. Please send requests for information to info@openEHR.org. Feedback should preferably be discussed on one of the appropriate mailing lists, openehr-technical@openehr.org or openehr-clinical@openehr.org.

2 Overview

The Common Reference Model comprises a number of packages containing concepts used in higher level *open*EHR models. It is illustrated in FIGURE 1.

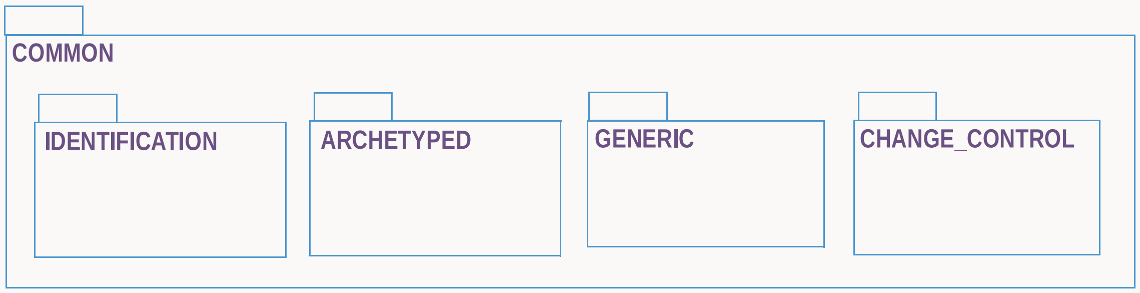

FIGURE 1 RM.COMMON Package Structure

The Identification package defines the semantics of all identifiers used in the *open*EHR models, including references to objects, identifiers of objects, and a representation of identifiers like the ISO OID and Microsoft Guid/ DCE UUID.

The Archetyped package described here is informed by a number of design principles, centred on the concept of “two-level” modelling. These principles are described in detail [1], and discussed with respect to the EHR [2].

The Generic package contains classes representing concepts which are generic across the domain.

The Change Control package defines the generalised semantics of changes to a repository, such as an EHR, over time. Each item in such a repository is version controlled to allow the repository as a whole to be properly versioned in time. The semantics described are in response to medico-legal requirements defined in GEHR [9], and in the ISO Technical Specification 18308 [4]. Both of these requirements specifications mention specifically the version control of the health record.

3 RM.COMMON.IDENTIFICATION Package

3.1 Requirements

Identification of entities both in the real world and in information systems is a non-trivial problem. The scenarios for identification across systems in a health information environment include the following:

- real world identifiers such as social security numbers, veterans affairs ids etc can be recorded as required by health care facilities, enterprise policies, or legislation.
- identifiers for informational entities which represent real world entities or processes should be unique.
- it should be possible to determine if two identifiers refer to information entities which are linked to the same real world entity, even if instances of the information entities are maintained in different systems;
- versions or changes to real-world entity-linked informational entities (which may create new information instances) should be accounted for in two ways:
    - it should be possible to tell if two identifiers refer to distinct versions of the same informational entity in the same version tree;
    - it should not be possible to confuse same-named versions of informational entities maintained in multiple systems which purport to represent the same real world entity. E.g. there is no guarantee that two systems' "latest" version of the Person "Dr Jones" is the same.

  Medico-legal use of information relies on previous states of information being identifiable in some way.
- it should be possible for an entity in one system or service (such as the EHR) to refer to an entity in another system or service in such a way that:
    - the target of the reference is easily finable within the shared environment, and
    - the reference does is valid regardless of the physical architecture of servers and applications.

The following subsections describe some of the features and challenges of identification.

Identification of Real World Entities (RWEs)

Real world entities such as people, car engines, invoices, and appointments all have identifiers. Although many of these are designed to be unique within a jurisdiction, they are often not, due to data entry errors, bad design (ids which are too small or incorporate some non-unique characteristic of the identified entities), bad process (e.g. non-synchronised id issuing points); identity theft (e.g. via theft of documents of proof or hacking). In general, while some real world identifiers (RWIs) are "nearly unique", none can be guaranteed so.

Examples of RWE identifiers which are intended to be unique over large jurisdictions include:

- driver's licence id
- social security number
- passport number

The defining characteristic of many RWE ids appears to be that they continue to identify the entities in question, regardless of how they changes in time; for example a social security number does not change when someone changes their hair colour or even their gender. There may be a general principle whereby any RWE id in fact doesn't identify an individual entity so much as its passage in time and space.

In general it should be the case that if two RWE ids are equal, they refer to the same RWE.

Identification of Informational Entities (IEs)

As soon as information systems are used to record facts about RWEs, the situation becomes more complex because of the intangible nature of information. In particular:

- the same RWE can be represented simultaneously on more than one system ("spatial multiplicity");
- the same RWE may be represented by more than one "version" of the same IE in a system ("temporal multiplicity").

At first sight, it appears that there can also be purely informational entities, i.e. IEs which do not refer to any RWE, such as books, online-only documents and software. However, as soon as one considers an example it becomes clear that there is always a notional "definitive" or "authoritative" (i.e. trusted) version of every such entity. These entities can better be understood as "virtual RWEs". Thus it can still be said that multiple IEs may refer to any given RWE.

The underlying reason for the multiplicity of IEs is that "reality" - time and space - in computer systems is not continuous but discrete, and each "entity" is in fact just a snapshot of certain attribute values of a RWE.

If identifiers are assigned to IEs without regard to versions or duplicates, then no assertion can be made about the identified RWE when two IE ids are compared.

Referencing of Informational Entities

Within a distributed information environment, there is a need for entities not connected by direct references in the same memory space to be able to refer to each other. There are two competing requirements:

- that the separation of objects in a distributed computing environment not compromise the semantics of the model. At the limit, this mandates the use of proxy types which have the same abstract interface as the proxied type; i.e. the "static" approach of Corba.
- that different types of information can be managed relatively independently; for example EHR and demographic information can be managed by different groups in an organisation or community, each with at least some freedom to change implementation and model details.

3.2 Overview

The External package describes a model of references and identifiers for information entities only and is illustrated in FIGURE 2.

The class `OBJECT_ID` is an abstract model of identifiers of IEs. It is assumed *a priori* that there can in general be more than one IE referring to the same underlying real world entity (RWE), such as a person or invoice; this is due to the possible existence of multiple copies, and also multiple versions. An `OBJECT_ID` therefore explicitly includes an optional *version_id* attribute. The rule for versioning is that if any attribute value of the IE changes, the version attribute value should be updated, e.g. by incrementing a simple integer. The *version_id* attribute should be used for object identifiers whose targets change, such as demographic entities; it can usually be omitted for ids of things like terminol-

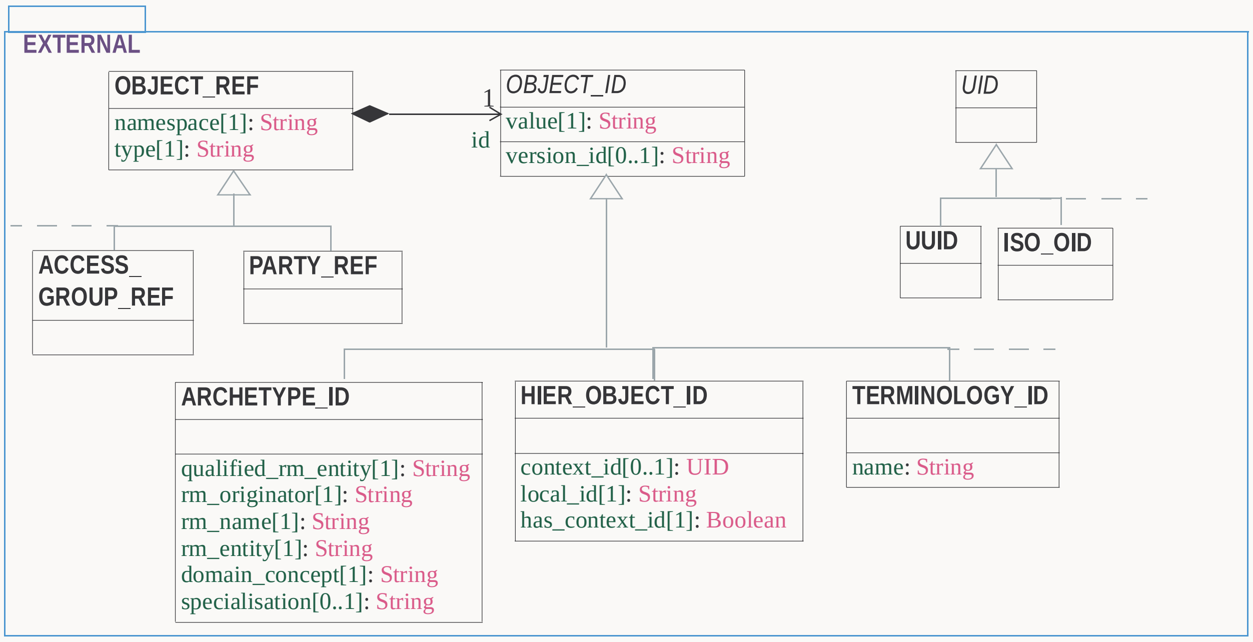

FIGURE 2 External Package

ogy codes, where the terminology obeys the rule that a given code never changes its meaning through all versions of the terminology (i.e. ICD10 code F40.0 will mean “Agoraphobia” for all time (in English)).

The subtype HIER_OBJECT_ID defines a hierarchical identifier model, along the lines of ISO Oids; it therefore includes the attributes *context_id* and *local_id*, to make up a complete, unique identifier. The *context_id* is optional, since it is possible for *local_id* values to exist in a single global namespace. When a HIER_OBJECT_ID has a *context_id*, it is of type UID, meaning it has the properties of a timeless unique object identifier. Subtypes of UID include the ISO_OID and DCE UUID types.

The other subtypes, ARCHETYPE_ID and TERMINOLOGY_ID define different kinds of identifier, the former being a multi-axial identifier for archetypes, and the latter being a globally unique single string identifier for terminologies.

All OBJECT_IDs are used as identifier attributes within the thing they identify. To *refer* to an identified object, an instance of the class OBJECT_REF is required. OBJECT_REF is provided as a means of distributed referencing, and includes the object namespace (typically 1:1 with some service, such as “terminology”) and type. The general principle of object references is to be able to refer to an object available in a particular namespace or service. Usually they are used to refer to objects in other services, such as a demographic entity from within an EHR, but they may be used to refer to local objects as well. The type may be the concrete type of the referred-to object (e.g. “GP”) or any proper ancestor (e.g. “PARTY”). The notion of object reference provided here is a compromise between the static binding notion of Corba (where each model is dependent on all the interface details of the classes in other models) and a purely dynamic referencing scheme, where the holder of a reference cannot even tell what type of object the reference points to.

3.3 Class Descriptions

3.3.1 OBJECT_REF Class

CLASS	OBJECT_REF	
Purpose	Class describing a reference to another object, which may exist locally or be maintained outside the current namespace, e.g. in another service. Services are usually external, e.g. available in a LAN (including on the same host) or the internet via Corba, SOAP, or some other distributed protocol. However, in small systems they may be part of the same executable as the data containing the Id.	
Attributes	Signature	Meaning
	id: `OBJECT_ID`	Globally unique id of an object, regardless of where it is stored.
	namespace: `String`	Namespace to which this identifier belongs in the local system context (and possibly in any other *open*EHR compliant environment) e.g. “terminology”, “demographic”. These names are not yet standardised. Legal values for the namespace are `“local” | “unknown” | “[a-zA-Z][a-zA-Z0-9_-:/&+?]*”`
	type: `String`	Name of the class of object to which this identifier type refers, e.g. “PARTY”, “PERSON”, “GUIDELINE” etc. These class names are from the relevant reference model. The type name “ANY” can be used to indicate that any type is accepted (e.g. if the type is unknown).
Invariant	***Id_exists***: id /= Void ***Namespace_exists***: namespace /= Void ***and then not*** namespace.empty ***Type_exists***: type /= Void ***and then not*** type.empty	

3.3.2 ACCESS_GROUP_REF Class

CLASS	ACCESS_GROUP_REF	
Purpose	Reference to access group in an access control service.	
Inherit	`OBJECT_REF`	
Functions	Signature	Meaning
Invariant	***namespace_validity***: namespace.is_equal(“access_control”) ***type_validity***: type.is_equal(“ACCESS_GROUP”)	

3.3.3 PARTY_REF Class

CLASS	PARTY_REF	
Purpose	Identifier for parties in a demographic service. There are typically a number of subtypes of the `PARTY` class, including `PERSON`, `ORGANISATION`, etc.	
Inherit	`OBJECT_REF`	
Functions	Signature	Meaning
Invariant	***namespace_validity***: namespace.is_equal(“demographic”)	

3.3.4 OBJECT_ID Class

CLASS	*OBJECT_ID (abstract)*	
Purpose	Ancestor class of identifiers of informational objects. Ids may be completely meaningless, in which case their only job is to refer to something, or may carry some information to do with the identified object.	
Use	Object_ids are used inside an object to identify that object. To identify another object, use an Object_ref.	
Attributes	Signature	Meaning
	value: `String`	The value of the id in the form defined below.
Functions	Signature	Meaning
	version_id: `String` ***ensure*** Result /= Void ***implies not*** Result.is_empty	Version of information pointed to by this ID, if versioning is supported.
Invariant	***Value_exists***: value /= Void ***and then not*** value.empty	

3.3.5 HIER_OBJECT_ID Class

CLASS	HIER_OBJECT_ID	
Purpose	Hierarchical identifiers.	
HL7	The HL7v3 II Data type.	
Functions	Signature	Meaning
	context_id: `UID`	The identifier of the conceptual namespace in which the object exists, within the identification scheme. May be Void.

CLASS	HIER_OBJECT_ID	
	has_context_id: `Boolean`	True if there is at least one "." in identifier before version part.
	local_id: `String` ***ensure*** Result /= Void ***and then not*** Result.empty	The local identifier of the object within the context.
Invariant		

3.3.5.1 Syntax

The syntax of the *value* attribute by default follows the following pattern:

```
    [ context_id "." ] local_id [ "(" version_id ")" ]
```

The syntax may be redefined in subtypes.

3.3.6 ARCHETYPE_ID Class

CLASS	ARCHETYPE_ID	
Purpose	Identifier for archetypes.	
Inherit	`OBJECT_ID`	
Functions	**Signature**	**Meaning**
	qualified_rm_entity: `String`	Globally qualified reference model entity, e.g. `"openehr-ehr_rm-entry"`.
	domain_concept: `String`	Name of the concept represented by this archetype, including specialisation, e.g. `"biochemistry result-choles-terol"`.
	rm_originator: `String` ***ensure*** Result /= Void ***and then not*** Result.is_empty	Organisation originating the reference model on which this archetype is based, e.g. `"openehr"`, `"cen"`, `"hl7"`.
	rm_name: `String` ***ensure*** Result /= Void ***and then not*** Result.is_empty	Name of the reference model, e.g. `"rim"`, `"ehr_rm"`, `"en13606"`.
	rm_entity: `String` ***ensure*** Result /= Void ***and then not*** Result.is_empty	Name of the ontological level within the reference model to which this archetype is targeted, e.g. for openEHR, "folder", "composition", "section", "entry".

CLASS	ARCHETYPE_ID	
	specialisation: `String` ***ensure*** Result /= Void ***implies not*** Result.is_empty	Name of specialisation of concept, if this archetype is a specialisation of another archetype, e.g. "cholesterol".
	local_id: `String` ***ensure then*** Result.is_equal(value)	
Invariant	***Qualified_rm_entity_valid***: qualified_rm_entity /= Void ***and then not*** qualified_rm_entity.is_empty ***Domain_concept_valid***: domain_concept /= Void ***and then not*** domain_concept.is_empty	

3.3.6.1 Archetype ID Syntax

Archetype ids obey the general pattern of object ids. They are defined in a single global namespace, hence the *context_id* attribute is always empty. The remaining part of the id is "multi-axial", meaning that each identifier instance denotes a single archetype within a multi-dimensional space. In this case, the space is essentially a versioned 3-dimensional space, with the dimensions being:

- reference model entity, i.e. target of archetype
- domain concept
- version

As with any multi-axial identifier, the underlying principle of an archetype id is that all parts of the id must be able to be considered immutable. This means that no variable characteristic of an archetype (e.g. accrediting authority, which might change due to later accreditation by another authority, or may be multiple) can be included in its identifier. The syntax of an `ARCHETYPE_ID` is as follows:

```
archetype_id: qualified_rm_entity '.' domain_concept '.' version_id

qualified_rm_entity: rm_originator '-' rm_name '-' rm_entity
rm_originator: NAME
rm_name: NAME
rm_entity: NAME

domain_concept: concept_name { '-' specialisation }*
concept_name: NAME
specialisation: NAME

version_id: 'v' NUMBER

NUMBER: [0-9]*
NAME: [a-z][a-z0-9()/%$#&]*
```

The field meanings are as follows:

rm_originator: id of organisation originating the reference model on which this archetype is based;

rm_name: id of the reference model on which the archetype is based;

rm_entity: ontological level in the reference model;

domain_concept: the domain concept name, including any specialisations;

version_id: numeric version identifier;

Examples of archetype identifiers include:

- `openehr-ehr_rm-section.physical_examination.v2`
- `openehr-ehr_rm-section.physical_examination-prenatal.v1`
- `hl7-rim-act.progress_note.v1`
- `openehr-ehr_rm-entry.progress_note-naturopathy.v2`

Archetypes can also be identified by other means, such as ISO oids.

3.3.7 TERMINOLOGY_ID Class

CLASS	**TERMINOLOGY_ID**	
Purpose	Identifier for terminologies such accessed via a terminology query service. In this class, the value attribute identifies the Terminology in the terminology service, e.g. “SNOMED-CT”. A terminology is assumed to be in a particular language, which must be explicitly specified. The value if the id attribute is the precise terminology id identifier, including actual release (i.e. actual “version”), local modifications etc; e.g. “ICPC2”	
Inherit	`OBJECT_ID`	
Functions	**Signature**	**Meaning**
	name: `String` ***ensure*** Result /= Void ***and then not*** Result.empty	Return the terminology id (which includes the “version” in some cases). Distinct names correspond to distinct (i.e. non-compatible) terminologies. Thus the names “ICD10AM” and “ICD10” refer to distinct terminologies.
	as_string: `String` ***ensure*** Result = key	
	as_canonical_string: `String`	Result = “<key>” + value + “</key>”
Invariants		

3.3.7.1 Identifier Syntax

The syntax of the *value* attribute is as follows:

```
name [ “(” version “)” ]
```

Examples of terminology identifiers include:

- `“snomed-ct”`
- `“ICD9(1999)”`

Versions should only be needed for those terminologies which break the rule that the thing being identified with a code loses or changes its meaning over versions of the terminology. This should not be the case for well known modern terminologies and ontologies, particularly those designed since the publication of Cimino's 'desiderata' [3] of which the principle of "concept permanance" is applicable here - "A concept's meaning cannot change and it cannot be deleted from the vocabulary". However, there maybe older terminologies, or specialised terminologies which may not have obeyed these rules, but which are still used; version ids should always be used for these.

3.3.8 UID Class

CLASS	*UID (abstract)*	
Purpose	Anstract parent of classes representing unique identifiers which identify information entities in a durable way. UIDs only ever identify one IE in time or space and are never re-used.	
HL7	The HL7v3 UID Data type.	
Attributes	Signature	Meaning
	value: `String`	The value of the id.
Invariant	***Value_exists***: value /= Void ***and then not*** value.empty	

3.3.9 ISO_OID Class

CLASS	ISO_OID	
Purpose	Model of ISO's Object Identifier (oid) as defined by the standard ISO/IEC 8824 . Oids are formed from integers separated by dots. Each non-leaf node in an Oid starting from the left corresponds to an assigning authority, and identifies that authority's namespace, inside which the remaining part of the identifier is locally unique.	
HL7	The HL7v3 OID Data type.	
Inherit	`UID`	
Functions	Signature	Meaning
Invariant		

3.3.10 UUID Class

CLASS	UUID	
Purpose	Model of the DCE Universal Unique Identifier or UUID which takes the form of hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by the Open Group, CDE 1.1 Remote Procedure Call specification, Appendix A.	
HL7	The HL7v3 UUID Data type.	
Inherit	`UID`	
Functions	Signature	Meaning
Invariant		

4 RM.COMMON.ARCHETYPED Package

4.1 Overview

The Archetype package includes the core types `LOCATABLE`, `ARCHETYPED`, and `LINK`. It is illustrated in FIGURE 3.

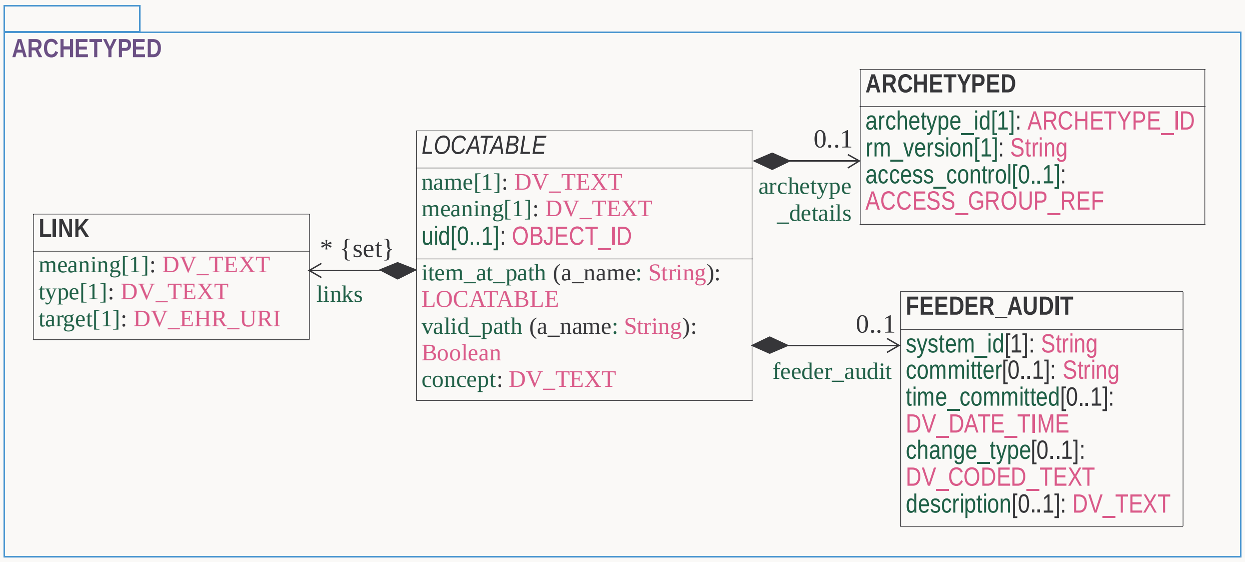

FIGURE 3 RM.COMMON.ARCHETYPED Package

4.1.1 The Root Class LOCATABLE

Every structural element in the EHR model inherits from the `LOCATABLE` class, ensuring it has both a runtime *name*, and a *meaning*. The *meaning* is the standardised, coded semantic name for a node, while the *name* attribute carries the runtime name. The name and meaning values are often the same, but can be different, for example in the "problem/SOAP" sections, where the name of an section at the problem level might be "diabetes", but its meaning (derived from its generating archetype) will be "problem". The default value for *names* should be assumed to be the value for *meaning*, unless explicitly set otherwise. `LOCATABLE` also provides the attribute *archetype_details*, which is non-Void for archetype root points in data.

`LOCATABLE` objects may also have a *uid*, typically implemented using an ISO Oid. In a given data composition, only those nodes which correspond to archetype root points should have a uid, since reliable paths can be generated to any point within the tree from a given root point. Thus, root points which might contain uids would normally be Compositions, Sections which are the root of a Section tree, and Entry objects; they could be finer grained nodes inside Entries if finer grained archetypes are used.

Currently the model does not formally mandate uids to be used, or to be used on any particular kind of node, despite the statements above, because there is not enough documented experience with using Oids for data node identification (particularly the computational costs of dereferencing, and the storage costs in otherwise 'small' data). More experience with real *open*EHR deployments is required before the correct formal semantics can be specified.

4.1.2 Feeder Systems

The data in any part of the EHR may be obtained from a feeder system, i.e. a source system which does not obey any of the EHR semantics defined by *open*EHR. In particular, there is no guarantee that the granularity of information recorded in the feeder system obeys the rules of Entries, Compositions, etc. As a consquence, feeder information might correspond to any level of information defined in the *open*EHR models. In order to be able to record feeder audit information correctly, the model has to be able to associate an audit trail with any granularity of object. For this reason, feeder audit information is attached to the LOCATABLE class via the *feeder_audit* attribute, even though it is preferable by design to have it attached to the equivalent of Compositions or at least the equivalent of archetype entities (i.e. Compositions, Section trees and Entries). Its usual usage is to attach it to the outermost object to which it applies. In other words, in most cases, during a legacy data conversion process, the entirety of a Composition needs only one FEEDER_AUDIT to document its origins. In exceptional cases, where feeder data comes in in near real time, e.g. from an ICU database, separate FEEDER_AUDIT objects may need to be generated for parts of a Composition; each commit in this situation will create a stack of versions of one Composition, with a growing number of FEEDER_AUDIT objects attached to internal data nodes, each documenting the last import of data.

The feeder audit information is included as part of the data of the Composition, rather than part of the audit trail of version committal, because it remains revelant throughout the versioning of a logical Composition, i.e. when a new version is created, the feeder information is retained as part of the current version to be seen and possibly modified, just as for the rest of its content. If the main part of the content is modified so drastically as to make the feeder audit irrelevant, it too can be removed.

A second consequence of feeder and legacy systems is that structural data items may need to be synthesised in order to create valid structures, even though the source system does not have them. For example, a system may have the equivalent data of Entries, but no Sections or other higher-level data items; these have to be synthesised during conversion. To indicate synthesis of a data node, a FEEDER_AUDIT instance is attached to the LOCATABLE in question, and its *change_type* set to “synthesised”.

4.2 Class Descriptions

4.2.1 Class LOCATABLE

CLASS	***LOCATABLE (abstract)***	
Purpose	Root structural class of all information models.	
GEHR	Name attribute in ARCHETYPED, *meaning* attribute in G1_PLAIN_TEXT.	
Synapses	Each record component includes a Synapses Object ID attribute to reference the Synapses Object (archetype) used as the basis for its construction. All record components include a name attribute intended for the same purpose as the *open*EHR equivalent.	
Attributes	**Signature**	**Meaning**
	uid: OBJECT_ID	Optional globally unique object identifier for root object of archetyped data structure.

CLASS	*LOCATABLE (abstract)*	
	meaning: `DV_TEXT`	Design-time name of this fragment taken from its generating archetype; used to build archetype paths. This value is therefore a "standardised" name for this EHR concept, taken from and mapping back to the archetype template according to which this instance has been constructed.
	name: `DV_TEXT`	Runtime name of this fragment, used to build runtime paths. This is the term provided via a clinical application or batch process to name this EHR construct: its retention in the EHR faithfully preserves the original label by which this entry was known to end users.
	archetype_details: `ARCHETYPED`	Details of archetyping used on this node.
	feeder_audit: `FEEDER_AUDIT`	Audit trail from non-*open*EHR system of original commit of information forming the content of this node, or from a conversion gateway which has synthesised this node.
	links: `Set <LINK>`	Links to other archetyped structures (data whose root object inherits from `ARCHETYPED`, such as `ENTRY`, `SECTION` and so on). Links may be to structures in other compositions.
Functions	**Signature**	**Meaning**
	is_archetype_root: `Boolean`	True if this node is the root of an archetyped structure.
	path_of_item (a_loc: `LOCATABLE`): `String`	The path to an item relative to the root of this archetyped structure.
	item_at_path (a_path: `String`): `LOCATABLE`	The item at a path (relative to this item).
	valid_path (a_path: `String`): `Boolean`	True if the path is valid with respect to the current item.
	concept: `DV_TEXT` ***require*** is_archetype_root	Clinical concept of the archetype as a whole (= derived from the 'meaning' of the root node)

CLASS	*LOCATABLE (abstract)*
Invariant	***Meaning_valid***: meaning /= Void ***Name_exists***: name /= Void ***Links_valid***: links /= Void ***implies not*** links.empty ***Archetyped_validity***: is_archetype_root ***xor*** archetype_details = Void

4.2.2 ARCHETYPED Class

CLASS	ARCHETYPED	
Purpose	Archetypes act as the configuration basis for the particular structures of instances defined by the reference model. To enable archetypes to be used to create valid data, key classes in the reference model act as “root” points for archetyping; accordingly, these classes have the archetype_details attribute set. An instance of the class `ARCHETYPED` contains the relevant archetype identification information, allowing generating archetypes to be matched up with data instances	
GEHR	`G1_ARCHETYPED`	
Synapses/ SynEx	The SynEx approach does not distinguish between multiple layers of archetypes; hence an ‘archetype’ covers all information in an entire composition. Consequently, there is only one place where archetype identifiers in the *open*EHR sense are used (at the top); all other archetype identifiers are equivalent to the *meaning* attribute from `LOCATABLE`. The Synapses ObjectID attribute provides a unique reference to each fine-grained element of an archetype, and is therefore also functionally equivalent to the *archetype_id* attribute at the root points in an *open*EHR structure.	
CEN	The 1999 pre-standard does not include any equivalent to the archetype concept. However each architectural component must include a reference to an entry in the relevant normative table in the Domain Termlist pre-standard (part 2), to provide a high-level semantic classification of the component. All Architectural components include a component name structure to specify its label: the source of possible values for such a label was not clearly defined. The 2003 revision of ENV 13606 explicitly includes archetype identification attributes in the class `RECORD_COMPONENT`.	
Attributes	Signature	Meaning
	archetype_id: `ARCHETYPE_ID`	Globally unique archetype identifier.
	access_control: `ACCESS_GROUP_REF`	The access control settings of this component.
	rm_version: `String`	Version of the *open*EHR reference model used to create this object.
Invariant	***archetype_id_exists***: archetype_id /= Void ***rm_version_exists***: rm_version /= Void ***and then not*** rm_version.empty	

4.2.3 LINK Class

CLASS	LINK	
Purpose	The LINK type defines a logical relationship between two items, such as two ENTRYs or an ENTRY and a COMPOSITION. Links can be used across compositions, and across EHRs. Links can potentially be used between interior (i.e. non archetype root) nodes, although this probably should be prevented in archetypes. Multiple LINKs can be attached to the root object of any archetyped structure to give the effect of a 1->N link	
Use	1:1 and 1:N relationships between archetyped content elements (e.g. ENTRYs) can be expressed by using one, or more than one, respectively, DV_LINKs. Chains of links can be used to see “problem threads” or other logical groupings of items.	
MisUse	Links should be between archetyped objects only, i.e. between objects representing complete domain concepts because relationships between sub-elements of whole concepts are not necessarily meaningful, and may be downright confusing. Sensible links only exist between whole ENTRYs, SECTIONs, COMPOSITIONs and so on.	
CEN	The Link Item class is a simplified form of the Synapses Link Item, permitting links to be established but with limited labelling and no representation for importance.	
Synapses	The Link Item class provides the means to link any arbitrary parts of a single EHR, for the overall linkage network to be labelled and revised, and for each direct link to be labelled explicitly. An importance attribute provides guidance on how links should be handled if only part of a linkage network is requested by a client process.	
GEHR	n/a	
HL7	The ACT_RELATIONSHIP class in some cases appears to correspond to LINK.	
Attributes	**Signature**	**Meaning**
	meaning: DV_TEXT	Used to describe the relationship, usually in clinical terms, such as “in response to” (the relationship between test results and an order), “follow-up to” and so on. Such relationships can represent any clinically meaningful connection between pieces of information. Value for meaning include those described in Annex C, ENV 13606 pt 2 [11] under the categories of “generic”, “documenting and reporting”, “organisational”, “clinical”, “circumstancial”, and “view management”.

CLASS	LINK	
	type: `DV_TEXT`	The *type* attribute is used to indicate a clinical or domain-level meaning for the kind of link, for example “problem” or “issue”. If type values are designed appropriately, they can be used by the requestor of EHR extracts to categorise links which must be followed and which can be broken when the extract is created.
	target: `DV_EHR_URI`	the logical “to” object in the link relation, as per the linguistic sense of the *meaning* attribute.
Functions	**Signature**	**Meaning**
Invariant	***meaning_exists***: meaning /= Void ***type_exists***: type /= Void ***target_exists***: target /= Void	

4.2.4 FEEDER_AUDIT Class

CLASS	FEEDER_AUDIT	
Purpose	Audit details for a feeder system.	
Attributes	**Signature**	**Meaning**
	system_id: `String`	Identity of the system where the item was originally committed.
	committer: `String`	Identity of party who committed the item.
	time_committed: `DV_DATE_TIME`	Time of committal of the item.
	change_type: `DV_CODED_TEXT`	Type of change, e.g. creation, correction, modification, synthesis etc. Coded using the *open*EHR Terminology “audit change type” group.
	description: `DV_TEXT`	Description of change from original system.
Invariants	***System_id_exists***: system_id /= Void ***and then not*** system_id.empty ***Committer_valid***: committer /= Void ***implies not*** committer.empty ***Change_type_valid***: change_type /= Void ***and then*** terminology(“openehr”).codes_for_group_name(“audit change type”, “en”).has(change_type.defining_code)	

5 RM.COMMON.GENERIC Package

5.1 Overview

The classes presented in this section are abstractions of concepts which are generic to the domain of health (and most likely other domains), such as “participation” and “attestation”. Here, “generic” means that the same model can be used, regardless of where they are contextually used in other models. The generic cluster is illustrated in FIGURE 4.

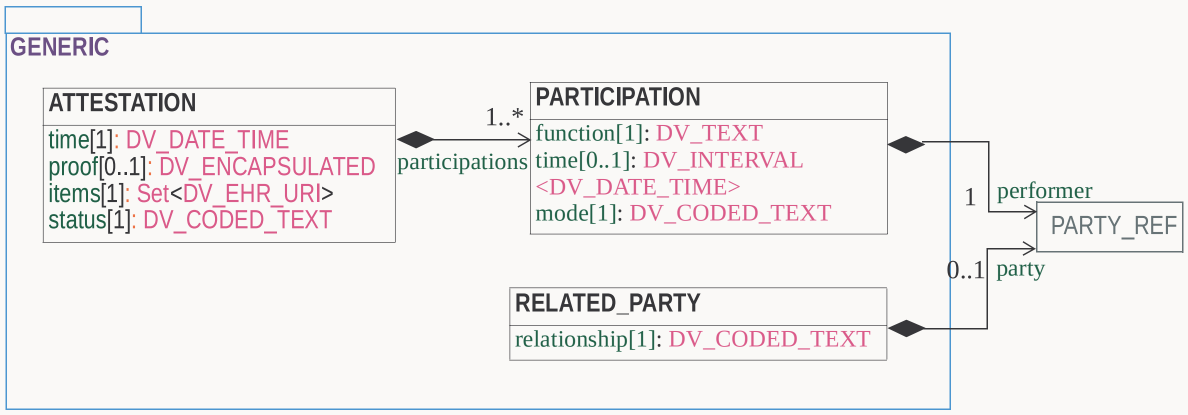

FIGURE 4 RM.COMMON.GENERIC Package

5.2 Participation

The concept of Participation is an abstraction of *the interaction between some Party and an activity.* In the *open*EHR reference models, participations are actually modelled in two ways. In situations where the kinds of participation are known and constant, they are modelled as a named attribute in the relevant reference model. For example, the *committer*:PARTY_REF attribute in CONTRIBUTION (inherited from AUDIT_DETAILS) models a participation in which the function is “committal”. Where the kind of participation is not known at design time, a generic PARTICIPATION class is used. This class refers to a Party, and records the function, time interval and mode of the participation. It can be used in any other reference model as required.

5.3 Attestation

Attestation is another concept which occurs commonly in health information. Here it is modelled as the combination of a number of participations, the time of attestation, a proof object and the list of identifiers of the attested items. At a minimum, the proof should be a digital certificate which binds:

- the identity of the parties
- the thing attested to, e.g. a statement like “Do you agree that the composition below is an accurate representation of the clinical session just completed?”, and potentially a hash or other compressed, encoded representation of the attested-to content
- the time
- appropriate digital signatures.

Such a certificate may be included in the record, or it may exist in some other place such as a notary service or similar. The use of the DV_ENCAPSULATED type for the *proof* attribute allows for either.

Normally the list of items should be a single Entry or Composition, but there is nothing stopping it including fine-grained items, even though separate attestation of such items does not appear to be commensurate with good clinical information design or process.

The *status* attribute is used to indicate whether the attestation has occurred, or is planned - the latter may be optional or mandatory. It is coded using the *open*EHR Terminology group “attestation status”.

5.3.1 Related Parties

The `RELATED_PARTY` class models the combination of a party reference and a relationship. It is intended to be used in any situation where the identity of the party may or may not be known, but the relationship to some other party is required - typically this will be a family relationship.

5.4 Class Descriptions

5.4.1 PARTICIPATION Class

CLASS	**PARTICIPATION**	
Purpose	Model of a participation of a Party (any Actor or Role) in an activity.	
Use	Used to represent any participation of a Party in some activity, which is not explicitly in the model, e.g. assisting nurse. Can be used to record past or future participations.	
Misuse	Should not be used in place of more permanent relationships between demographic entities.	
HL7	RIM Participation class.	
Attributes	**Signature**	**Meaning**
	performer: `PARTY_REF`	The party participating in the activity.
	function: `DV_CODED_TEXT`	The function of the Party in this participation (note that a given party might participate in more than one way in a particular activity). This attribute should be coded, but probably cannot be limited to the HL7:Participation-Function vocabulary, since it is too limited and hospital-oriented.
	mode: `DV_CODED_TEXT`	The mode of the performer / activity interaction, e.g. present, by telephone, by email etc.
	time: `DV_INTERVAL <DV_DATE_TIME>`	The time interval during which the participation took place, if it is used in an observational context (i.e. recording facts about the past); or the intended time interval of the participation when used in future contexts, such as EHR Instructions.

CLASS	PARTICIPATION
Invariant	***Function_exists***: function /= Void ***Mode_valid***: terminology("openehr").codes_for_group_name("participation modes", "en").has(mode) ***Performer_exists***: performer /= Void

5.4.2 ATTESTATION Class

CLASS	ATTESTATION	
Purpose	Record of one or more Parties attesting something.	
Attributes	Signature	Meaning
	participations: `List <PARTICIPATION>`	Pariticipations in this attestation, e.g. witness, primary authority etc.
	time: `DV_DATE_TIME`	Time at which attestation was made.
	proof: `DV_ENCAPSULATED`	Proof of attestation.
	items: `Set <DV_EHR_URI>`	Items attested. Although not recommended, these may include fine-grained items which have been attested in some other system. Otherwise it is assumed to be for the entire `VERSION` with which it is associated.
	status: `DV_CODED_TEXT`	Status of this attestation. Coded by the *open*EHR Terminology group "attestation status".
Invariants	***Participations_validity***: participations /= Void ***and then not*** participations.empty ***Time_exists***: time /= Void ***Items_validity***: items /= Void ***and then not*** items.is_empty ***Status_validity***: status /= Void ***and then*** terminology("openehr").codes_for_group_name("attestation status", "en").has(status.defining_code)	

5.4.3 RELATED_PARTY Class

CLASS	RELATED_PARTY	
Purpose	Party and relationship of the party.	
Attributes	Signature	Meaning
	party: `PARTY_REF`	Id of Party

CLASS	RELATED_PARTY	
	relationship: DV_CODED_TEXT	Relationship of subject of this ENTRY to the subject of the record. May be coded. If it is the patient, coded as “self”.
Invariants	***Relationship_valid***: relationship /= Void ***and then*** terminology(“openehr”).codes_for_group_name(“related party relationship”, “en”).has(relationship.defining_code)	

6 RM.COMMON.CHANGE_CONTROL Package

6.1 Overview

In various *open*EHR reference models, the semantics of formal change control are required. There are two architectural aspects of managing changes to data. The first is the concept of a complex information object, being versioned in time, meaning that its creation and all subsequent modifications cause new "versions" to be created, rather than literally overwriting the existing data. Each version includes an audit trail, typically containing the identity of a user, the date/time of the change, and a reason for the change. The second aspect recognises that repositories are made up of complex information objects, and that changes are not in fact just made to individual objects, but to the respository itself. Any change by a user may in fact change more than one versioned object in the repository, and the set of such changes constitutes the logical unit of change to the repository, taking it from one valid state to the next.

These concepts are well-known, under the title "configuration management", and are used as the basis for most software and other change management systems, including numerous products on the market today.

The following sections describe the configuration management paradigm in more detail, and explain how it relates to the *open*EHR reference models, in particular, the model for the EHR.

6.2 The Configuration Management Paradigm

The "configuration management" (CM) paradigm is well-known in software engineering, and has its own IEEE standards. CM is about managed control of changes to a repository of items (formally called "configuration items" or CIs), and is relevant to any logical repository of distinct information items which changes in time. In health information systems, at least two types of information require such management: electronic health records, and demographic information. In most analyses in the past, the need for change management has been expressed in terms of specific requirements for audit trailing of changes, availability of previous states of the repository and so on. Here, we aim to provide a formal, general-purpose model for change control, and show how it applies to health information.

6.2.1 Organisation of the Repository

The general organisation of a repository of complex information items such as a software repository, or the EHR consists of the following:

- a number of distinct information items, or *configuration items*, each of which is uniquely identified, and may have any amount of internal complexity;
- optionally, a directory system of some kind, in which the configurations items are organised.

Thus, in a software or document repository, the CIs are files arranged in the directories of the file system; in an EHR based on the GEHR or CEN models, they are Compositions arranged in a Folder structure. Contributions are made to the repository by users. This general abstraction is visualised in FIGURE 5.

6.2.2 Change Management

As implied earlier, change doesn't occur to CIs in isolation, but to the repository as a whole. Possible types of change include:

- creation of a new CI;

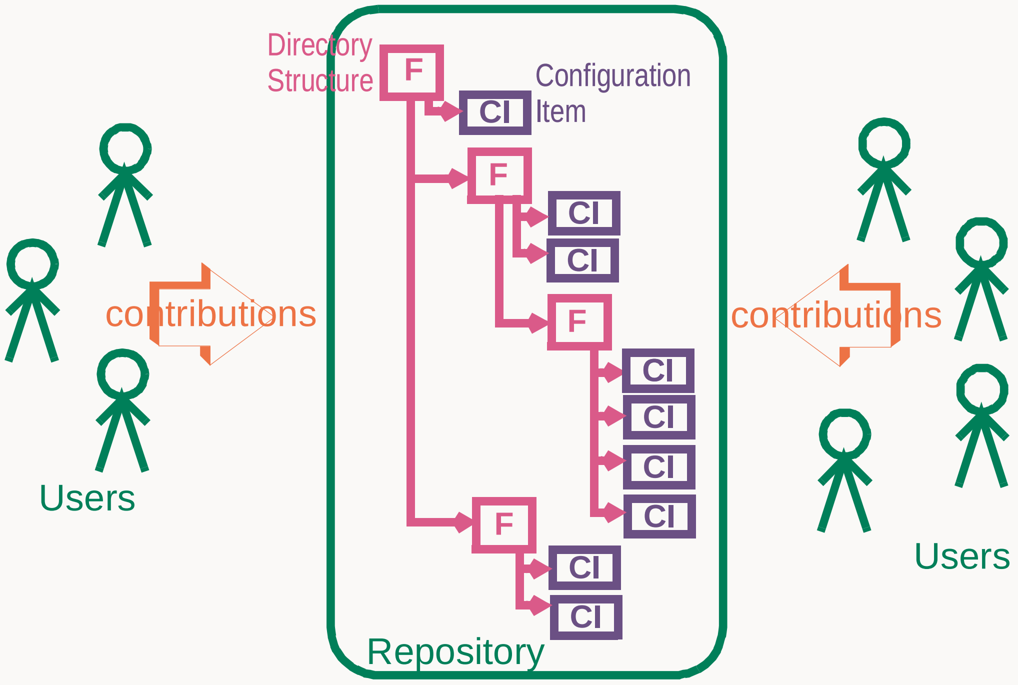

FIGURE 5 General Structure of a Controlled Repository

- removal of a CI;
- modification of a CI;
- creation of, change to or deletion of part of the directory structure;
- moving of a CI to another location in the directory structure.

The goal of configuration management is to ensure the following:

- the repository is always in a valid state;
- any previous state of the repository can be reconstructed;
- all changes are audit-trailed.

6.2.3 Changes in Time

Properly managing changes to the repository requires two mechanisms. The first, *version control*, is used to manage versions of each CI, and of the directory structure if there is one. The second is the concept of the "change-set", or what we will call a *contribution*. This is the *set* of changes to individual CIs (and the directory structure) made by a user as part of some logical change. For example, in a document repository, the logical change might be an update to a document that consists of multiple files (CIs). There is one contribution, consisting of changes to the document file CIs, to the repository. In the EHR, a contribution might consist of changes to more than one Composition, and possibly to the organising Folder structure.

A typical sequence of changes to a repository is illustrated below. FIGURE 6 shows a notional repository containing a number of CIs in an initial state (note that the directory tree is not shown for the sake of simplicity).

The repository after four contributions is shown in FIGURE 7 (where each contribution is indicated by a blue oval). As each contribution is made, the repository is changed in some way. The first brings into existing a new CI, and modifies three others (changes indicated by the 'C' triangles). The second

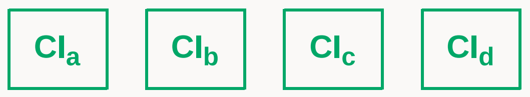

FIGURE 6 Initial State of Repository

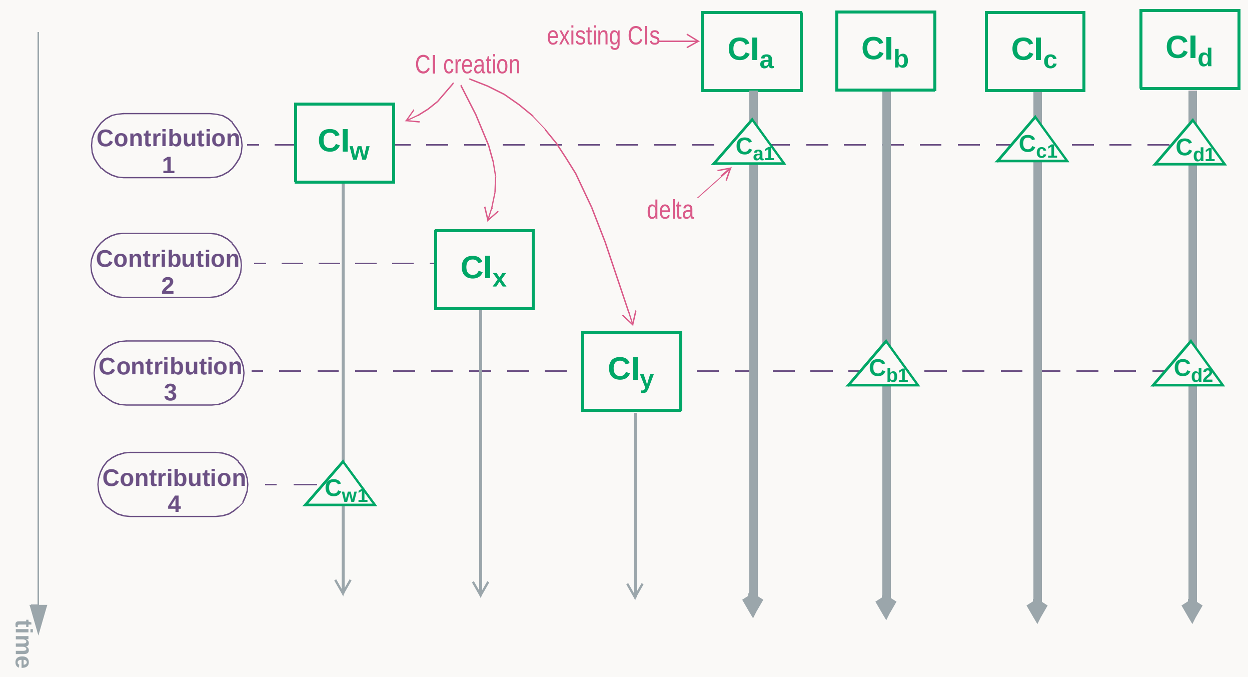

FIGURE 7 Contributions to the Repository (delta form)

contribution causes the creation of a new CI only. The third causes a creation as well as two changes, while the fourth causes only a change. (Again, changes to the folder structure are not shown here).

One nuance which should be pointed out is that, in FIGURE 7, contributions are shown as if they are literally a set of deltas, i.e. exactly the changes which occur to the record. Thus, the first contribution is the set {CI_w, C_{a1}, C_{c1}, C_{d1}} and so on. Whether this is exactly true depends on the construction of applications. In some situations, some CIs may be updated by the user viewing the current list and entering just the changes - the situation shown in FIGURE 7; in others, the system may provide the current state of these CIs for editing by the user, and submit the updated versions, as shown in FIGURE 8. Some applications may do both, depending on which CI is being updated. The internal versioning implementation may or may not generate deltas as a way of efficient storage.

For our purposes here, we consider a contribution as being the logical set of CIs changed or created at one time, as implied by FIGURE 8.

6.2.4 General Model of a Change-controlled Repository

FIGURE 9 shows an abstract model of a change-controlled repository, which consists of:

- version-controlled configuration items - instances of `VERSION_REPOSITORY<T>`;
- `CONTRIBUTIONs`;
- an optional directory system of of folders. If folders are used, the folder structure must also be versioned as a unit.

The actual type of links between the controlled repository and the other entities might vary - in some cases it might be composition, in others aggregation; cardinalities might also vary. FIGURE 9 there-

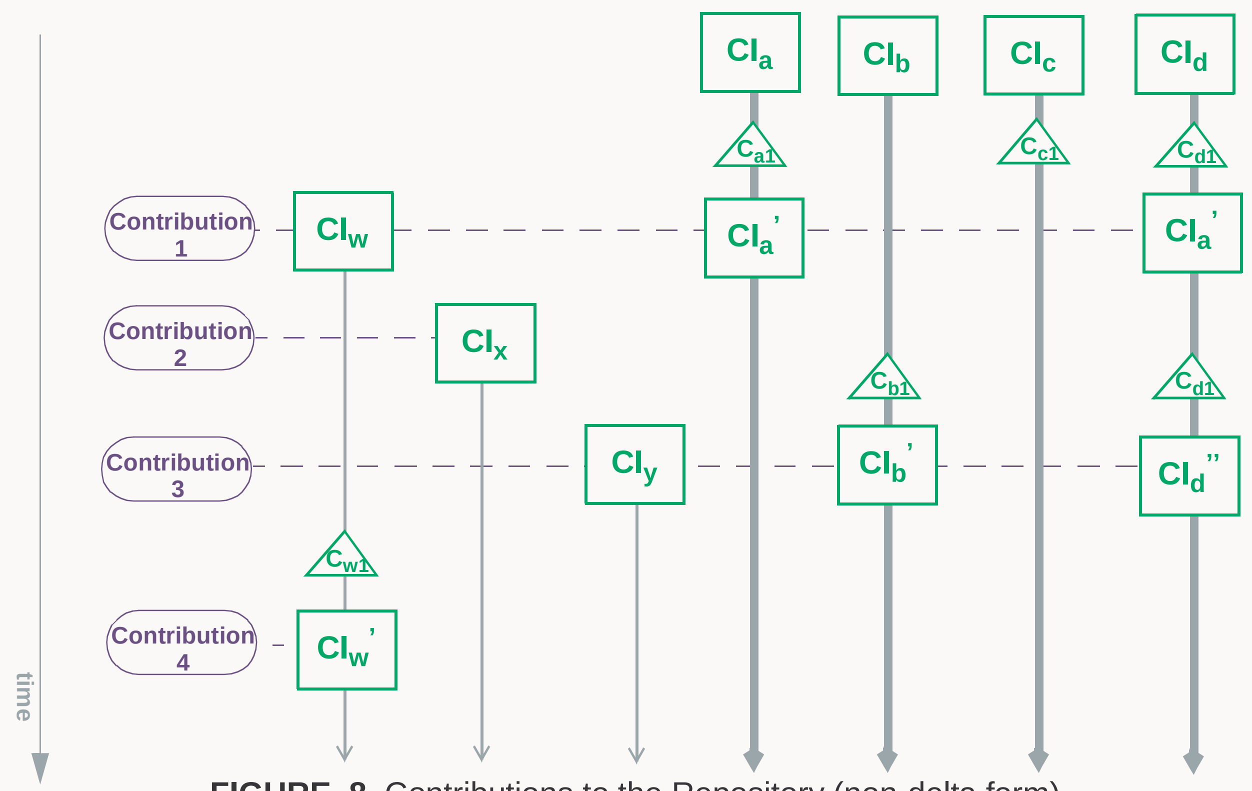

FIGURE 8 Contributions to the Repository (non-delta form)

fore provides a guide to the definition of actual controlled repositories, such as an EHR, rather than a formal specification for them.

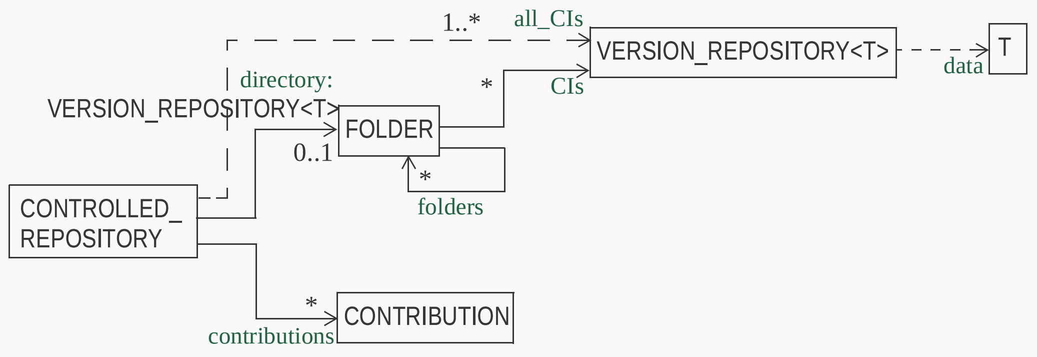

FIGURE 9 Informal Model of Change-controlled Repository

6.3 Formal Model

6.3.1 Versioned Items

FIGURE 10 illustrates a formal model of a version repository. In this model, the class VERSION_REPOSITORY<T> provides the versioning facilities for one CI, such as an EHR Composition, or a Party in a demographic system. Each version is an instance of the class VERSION<T>, which combines the data being versioned, the audit trail, and any attestations applied to the version. Both VERSION_REPOSITORY<T> and VERSION<T> are generic classes, with the generic parameter type T being the type of the data; ensuring that all versions in a given VERSION_REPOSITORY are of the same type, such as ENTRY or FOLDER, and that the repository itself is properly typed. Each VERSION_REPOSITORY has a unique identifier, its *uid* attribute, and a reference to the owning object

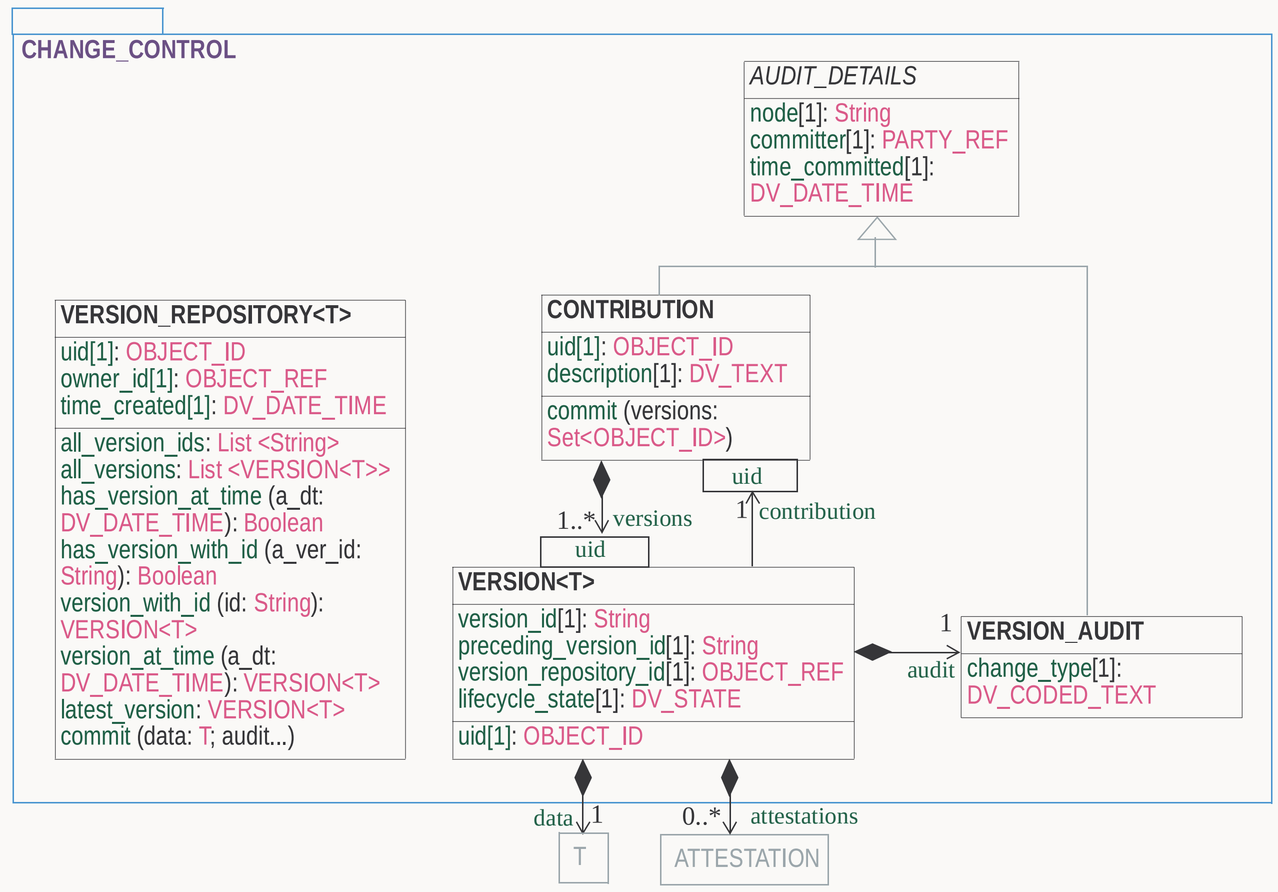
CHANGE_CONTROL
AUDIT_DETAILS
node[1]: String
committer[1]: PARTY_REF
time_committed[1]: DV_DATE_TIME
VERSION_REPOSITORY<T>
uid[1]: OBJECT_ID
owner_id[1]: OBJECT_REF
time_created[1]: DV_DATE_TIME
all_version_ids: List <String>
all_versions: List <VERSION<T>>
has_version_at_time (a_dt: DV_DATE_TIME): Boolean
has_version_with_id (a_ver_id: String): Boolean
version_with_id (id: String): VERSION<T>
version_at_time (a_dt: DV_DATE_TIME): VERSION<T>
latest_version: VERSION<T>
commit (data: T; audit...)
CONTRIBUTION
uid[1]: OBJECT_ID
description[1]: DV_TEXT
commit (versions: Set<OBJECT_ID>)
uid
1 contribution
1..* versions
uid
VERSION<T>
version_id[1]: String
preceding_version_id[1]: String
version_repository_id[1]: OBJECT_REF
lifecycle_state[1]: DV_STATE
uid[1]: OBJECT_ID
1
audit
VERSION_AUDIT
change_type[1]: DV_CODED_TEXT
data 1
0..* attestations
T
ATTESTATION

FIGURE 10 RM.COMMON.CHANGE_CONTROL Package

- the *owner_id* attribute. The latter helps ensure that in storage systems, versioned objects are always correctly allocated to their enclosing repository, such as an EHR.

The *audit* attribute in VERSION<T>, of type VERSION_AUDIT, includes of a set of attributes which form an audit trail, namely *node*, *committer*, *time_committed*, and *reason*, and provides a link to the contribution responsible for the version. It . The first three of these attribute values always have values identical with the CONTRIBUTION instance for a given contribution, i.e. *they are copied in*. This is done to enable sharing of versioned entities independently of which contributions they were part of.

The *attestations* attribute allows attestations to be associated with the data in the version. Attestations can be used as required by enterprise processes or legislation, and indicate who and when the item in question was attested. A digital “proof” is also required, although no assumption is made about the form of such proof.

6.3.2 Version Lifecycle

Versioned content has a lifecycle state associated with it, modelled using the VERSION.*lifecycle_state* attribute. Typically the lifecycle would follow the trajectory “draft” -> “active” -> “inactive”. No particular state machine is mandated here, although the state attribute is coded from the *open*EHR Terminology “version lifecycle state” group. Recording the state in VERSION enables two common scenarios to be faithfully represented.

The first is a user's need to save something in 'draft' state, e.g. because they have run out of time to finish writing part of the Composition, or due to an emergency. In hospitals this may well be a common occurrence. Such draft Compositions cannot be saved locally on the client machine, since this represents a security risk (a small client-side database would be much easier to hack into than a secure server). They must therefore be persisted on the server, either in the actual EHR, or in a 'holding bay' which was recognised as not being part of the EHR proper. Either way, the author would have to explicitly retrieve the Composition(s) and after further work or review, 'promote' them into the EHR as 'active' Compositions; alternatively, they might decide to throw them away.

The second scenario is the need to be able to represent states like 'active', 'inactive' and 'deleted' of Compositions in a system. (The 'deleted' state is of course only logical - in a medico-legally safe record, nothing can be actually deleted).

In both scenarios, it should be possible to change the status of a Composition without actually changing the Composition itself - i.e. without creating a new version. This requirement prevents storing it in Composition itself; it is thus stored in the more logically correct place, `VERSION`. `VERSION`.*lifecycle_state* might in some systems be linked to the state of attestations (described below) which are also attached to `VERSION` objects.

6.3.3 Attestation of Versions

Scenarios relating to attestation may cause attestations to be created at different times with respect to the committal of data to the EHR, as follows:

- *at committal*: highly sensitive information is to be added to the EHR, e.g. recording the fact of sectioning of a patient under the mental health act, diagnosis of a fatal disease etc. In this case, attestation is added at committal to the EHR;
- *post-committal*: a data-entry person e.g. a secretary, transcriptionist or student is responsible for entering the data, including routine things such as referrals, discharge summaries etc, which need to be verified by the relevant clinician; this may occur after committal to the EHR in some cases, leading to the temporary presence of entries "awaiting attestation" in the record.
- *not required*: attestation not required - at many locations, many types of EHR additions do not require any special attestation at all, and can be committed to the EHR by a wider range of personnel.

As a result of these requirements, the model allows any number of attestations (from 0 to many) to be associated with each version of a versioned object. Attestations are considered to be neither part of the content, nor part of the audit information, but an external artifact which refers in to versions of versioned items. Attestations can be added at any time.

The class `CONTRIBUTION` defines the common audit information for the set of versions added to the repository due to a given contribution as well as a *description* of the contribution as a whole. `CONTRIBUTIONS` refer to their member `VERSION` objects via `OBJECT_IDs`; similarly, the audit object of any `VERSIONABLE` refers to its creating `CONTRIBUTION` using an `OBJECT_IDs` reference.

These classes can be used to provide versioning and contributions in repositories such as an EHR, or a demographic repository. In the EHR reference model for example, to obtain a versioned Composition, the type `VERSION_REPOSITORY<COMPOSITION>` is defined.

6.4 Class Descriptions

6.4.1 VERSION_REPOSITORY Class

CLASS	VERSION_REPOSITORY<T>	
Purpose	Version control abstraction, defining semantics for versioning one complex object.	
Attributes	Signature	Meaning
	uid: `OBJECT_ID`	Unique identifier of this version repository.
	owner_id: `OBJECT_REF`	Reference to object to which this versioned repository belongs, e.g. the id of the containing EHR.
	time_created: `DV_DATE_TIME`	Time of initial creation of this versioned object.
Functions	Signature	Meaning
	all_versions: `List <VERSION<T>>`	Return a list of all versions in this object.
	all_version_ids: `List <String>`	Return a list of ids of all versions in this object.
	version_count: `Integer`	Return the total number of versions in this object
	has_version_id (an_id: `String`): `Boolean` ***require*** an_id /= Void ***and then not*** an_id.is_empty	True if a version with id ‘an_id’ exists.
	has_version_at_time (a_time:`DV_DATE_TIME`): `Boolean` ***require*** a_time /= Void	True if a version for time ‘a_time’ exists.
	version_with_id (an_id:`String`): `VERSION<T>` ***require*** has_version_with_id(an_id)	Return the version with id ‘an_id’.

CLASS	VERSION_REPOSITORY<T>	
	version_at_time (a_time:`DV_DATE_TIME`): `VERSION<T>` ***require*** has_version_at_time(a_time)	Return the version for time ‘a_time’.
	latest_version: `VERSION<T>`	Return the latest version.
	commit (an_audit: `VERSION_AUDIT`; a_version: `T`) ***require*** an_audit /= Void a_version /= Void	Add a new version.
Invariant	***uid_exists***: uid /= Void ***owner_id_valid:*** owner_id /= Void ***time_created_exists***: time_created /= Void ***versions_exists***: version_count >= 1	

6.4.2 AUDIT_DETAILS Class

CLASS	*AUDIT_DETAILS (abstract)*	
Purpose	The set of attributes required to document a new version of something. This class can be inherited or used in a client/supplier relationship to provide audit trail details to another class.	
Attributes	Signature	Meaning
	node: `String`	Identity of the node where the item was committed.
	committer: `PARTY_REF`	Identity of party who committed the item.
	time_committed: `DV_DATE_TIME`	Time of committal of the item.
Invariants	***Node_exists***: node /= Void ***and then not*** node.empty ***Committer_exists***: committer /= Void ***Time_committed_exists***: time_committed /= Void	

6.4.3 VERSION Class

CLASS	VERSION<T>	
Purpose	Versionable objects, with an audit trail containing details of change.	
Attributes	Signature	Meaning

CLASS	VERSION<T>	
	data: `T`	The data being versioned.
	attestations: `List <ATTESTATION>`	Set of attestations relating this version.
	audit: `VERSION_AUDIT`	Audit trail of this version.
	version_id: `String`	Unique identifier of this version.
	preceding_version_id: `String`	Unique identifier of the version on which this version was based. May be the pseudo-version "first".
	version_repository_id: `OBJECT_REF`	A copy of the uid of the version repository to which this version was added.
	contribution: `OBJECT_REF`	Contribution in which this version was added.
	lifecycle_state: `DV_STATE`	Lifecycle state of the content item in this version.
Functions	**Signature**	**Meaning**
	uid: `OBJECT_ID`	Unique identifier of this version, derived from version repository id and version id.
Invariant	***version_id_exists***: version_id /= Void ***and then not*** version_id.is_empty ***preceding_version_id_exists***: preceding_version_id /= Void ***and then not*** preceding_version_id.is_empty ***version_repository_id_exists***: version_repository_id /= Void ***lifecycle_state_valid***: lifecycle_state /= Void ***and then*** terminology("openehr").codes_for_group_name("version lifecycle state", "en").has(lifecycle_state.value.defining_code) ***audit_exists***: audit /= Void ***attestations_valid***: attestations /= Void ***implies not*** attestations.is_empty ***Contribution_exists***: contribution /= Void ***uid_valid***: uid /= Void ***and*** uid.version_id.is_equal(version_id)	

6.4.4 VERSION_AUDIT Class

CLASS	VERSION_AUDIT
Purpose	Audit trail data for a version of something. Points back to a contribution object, which contains the audit details common to all versions in a contribution.
Synapses	Composition class
GEHR	`G1_COMMIT_AUDIT`

CLASS	VERSION_AUDIT	
Inherit	`AUDIT_DETAILS`	
Attributes	Signature	Meaning
	change_type: `DV_CODED_TEXT`	Type of change. Coded using the *open*EHR Terminology “audit change type” group.
Invariant	***Change_type_exists***: change_type /= Void ***and then*** terminology(“openehr”).codes_for_group_name(“audit change type”, “en”).has(change_type.defining_code)	

6.4.5 CONTRIBUTION Class

CLASS	CONTRIBUTION	
Purpose	Documents a contribution of one or more versions added to a change-controlled repository.	
Inherit	`AUDIT_DETAILS`	
Attributes	Signature	Meaning
	uid: `OBJECT_ID`	Unique identifier for this contribution.
	description: `DV_TEXT`	Reason for committal.
	versions: `Set<OBJECT_REF>`	Set of references to versions causing changes to this EHR. Each contribution contains a list of versions, which may include paths pointing to any number of `VERSIONABLE` items, i.e. items of type `COMPOSITION` and `FOLDER`.
Invariants	***uid_exists***: uid /= Void ***Description_exists***: description /= Void ***Versions_valid***: versions /= Void ***and then not*** versions.empty	

6.5 Transaction Semantics of Contributions

In terms of database management, Contributions are considered as nested transactions. The attempt to commit a Contribution should only succeed if each `VERSION` instance in the Contribution is committed successfully. Failure to commit any of the member instances should cause failure of the Contribution.

A References

A.1 General

1 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. See http://www.deepthought.com.au/it/archetypes.html.

2 Beale T *et al*. *Design Principles for the EHR*. See http://www.openEHR.org.

3 Cimino J J. *Desiderata for Controlled Medical vocabularies in the Twenty-First Century*. IMIA WG6 Conference, Jacksonville, Florida, Jan 19-22, 1997.

4 Schloeffel P. (Editor). Requirements for an Electronic Health Record Reference Architecture. International Standards Organisation, Australia; Feb 2002; ISO TC 215/SC N; ISO/WD 18308.

A.2 European Projects

5 Dixon R., Grubb P.A., Lloyd D., and Kalra D. *Consolidated List of Requirements. EHCR Support Action Deliverable 1.4*. European Commission DGXIII, Brussels; May 200159pp Available from http://www.chime.ucl.ac.uk/HealthI/EHCR-SupA/del1-4v1_3.PDF.

6 Dixon R, Grubb P, Lloyd D. *EHCR Support Action Deliverable 3.5: "Final Recommendations to CEN for future work"*. Oct 2000. Available at http://www.chime.ucl.ac.uk/HealthI/EHCR-SupA/documents.htm.

7 Dixon R, Grubb P, Lloyd D. *EHCR Support Action Deliverable 2.4 "Guidelines on Interpretation and implementation of CEN EHCRA"*. Oct 2000. Available at http://www.chime.ucl.ac.uk/HealthI/EHCR-SupA/documents.htm.

8 Ingram D. *The Good European Health Record Project*. Laires, Laderia Christensen, Eds. Health in the New Communications Age. Amsterdam: IOS Press; 1995; pp. 66-74.

9 *Deliverable 19,20,24: GEHR Architecture*. GEHR Project 30/6/1995

A.3 CEN

10 ENV 13606-1 - *Electronic healthcare record communication - Part 1: Extended architecture*. CEN/ TC 251 Health Informatics Technical Committee.

11 ENV 13606-2 - *Electronic healthcare record communication - Part 2: Domain term list*. CEN/ TC 251 Health Informatics Technical Committee.

12 ENV 13606-4 - *Electronic Healthcare Record Communication standard Part 4: Messages for the exchange of information*. CEN/ TC 251 Health Informatics Technical Committee.

A.4 OMG

13 CORBAmed document: *Person Identification Service*. (March 1999). (Authors?)

14 CORBAmed document: *Lexicon Query Service*. (March 1999). (Authors?)

A.5 Software Engineering

15 Meyer B. *Object-oriented Software Construction*, 2nd Ed.
Prentice Hall 1997

16 Fowler M. *Analysis Patterns: Reusable Object Models.* Addison Wesley 1997

17 Fowler M, Scott K. *UML Distilled (2nd Ed.).* Addison Wesley Longman 2000

A.6 Resources

18 IANA - http://www.iana.org/.

END OF DOCUMENT