REFERENCE MODEL

The *open*EHR Common Information Model

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

Revision: 2.0

Pages: 63

1. Ocean Informatics Australia

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

Copyright Notice

Amendment Record

Issue	Details	Raiser	Completed
RELEASE 1.0			
2.0	**CR-000147**: Make DIRECTORY Re-usable. Add new directory package. **CR-000162**. Allow party identifiers when no demographic data. **CR-000167**. Add AUTHORED_RESOURCE class. **CR-000179**. Move AUDIT_DETAILS to generic package; add REVISION_HISTORY. **CR-000182**: Rationalise VERSION.*lifecycle_state* and ATTESTATION.*status* **CR-000065**. Add Revision History to change control package. **CR-000187**: Correct modelling errors in DIRECTORY class and rename. **CR-000163**: Add identifiers to FEEDER_AUDIT for originating and gateway systems. **CR-000165.** Clarify use of *system_id* in FEEDER_AUDIT and AUDIT_DETAILS. **CR-000190**. Rename VERSION_REPOSITORY to VERSIONED_OBJECT . **CR-000161**. Support distributed versioning. Additions to change_control package.	T Beale S Heard H Frankel T Beale T Beale C Ma D Kalra T Beale T Beale H Frankel H Frankel T Beale H Frankel, T Beale	02 Feb 2006
RELEASE 0.96			
1.6.2	**CR-000159**. Improve explanation of use of ATTESTATION in change_control package.	T Beale	10 Jun 2005
RELEASE 0.95			
1.6.1	**CR-000048**. Pre-release review of documents. Fixed UML in Fig 8 informal model of version control.	D Lloyd	22 Feb 2005
1.6	**CR-000108**. Minor changes to change_control package. **CR-000024**. Revert *meaning* to STRING and rename as *archetype_node_id*. **CR-000097**. Correct errors in version diagrams in Common model. **CR-000099**. PARTICIPATION.*function* type in diagram not in sync with spec. **CR-000116**. Add PARTICIPATION.*function* vocabulary and invariant. **CR-000118**. Make package names lower case. Improve presentation of identification section; move some text to data types IM document, basic package. **CR-000111**. Move Identification Package to Support	T Beale S Heard T Beale Ken Thompson R Shackel (DSTC) T Beale T Beale DSTC	10 Dec 2004
RELEASE 0.9			

Issue	Details	Raiser	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	Raiser	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 Information Model
- The *open*EHR Data Types Information Model
- The *open*EHR Data Structures Information Model

1.3 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/rm/common_im.pdf`.

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

New versions are announced on `openehr-announce@openehr.org`.

Blue text indicates sections under active development.

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 provided on the mailing list `openehr-technical@openehr.org`, or by private email.

1.5 Conformance

Conformance of a data or software artifact to an *open*EHR Reference Model specification is determined by a formal test of that artifact against the relevant *open*EHR Implementation Technology Specification(s) (ITSs), such as an IDL interface or an XML-schema. Since ITSs are formal, automated derivations from the Reference Model, ITS conformance indicates RM conformance.

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

2 Overview

The `common` Reference Model comprises a number of packages containing abstract concepts and design patterns used in higher level *open*EHR models. It is illustrated in FIGURE 1.

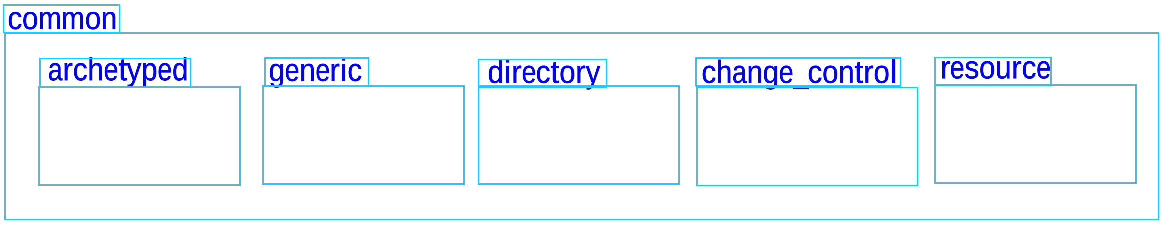

FIGURE 1 rm.common Package

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 in [1].

The `generic` package contains classes representing concepts which are generic across the domain, mostly to do with referencing demographic entities from within other data including Participation, Party_proxy, Attestation and so on.

The `directory` package provides a simple reusable abstraction of a versioned folder structure.

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.

The `resource` package defines semantics of an online authored resource, such as a document, and supports multiple language translations, descriptive meta-data and revision history.

3 Archetyped Package

3.1 Overview

The `archetyped` package includes the core types `LOCATABLE`, `ARCHETYPED`, and `LINK`. It is illustrated in FIGURE 2.

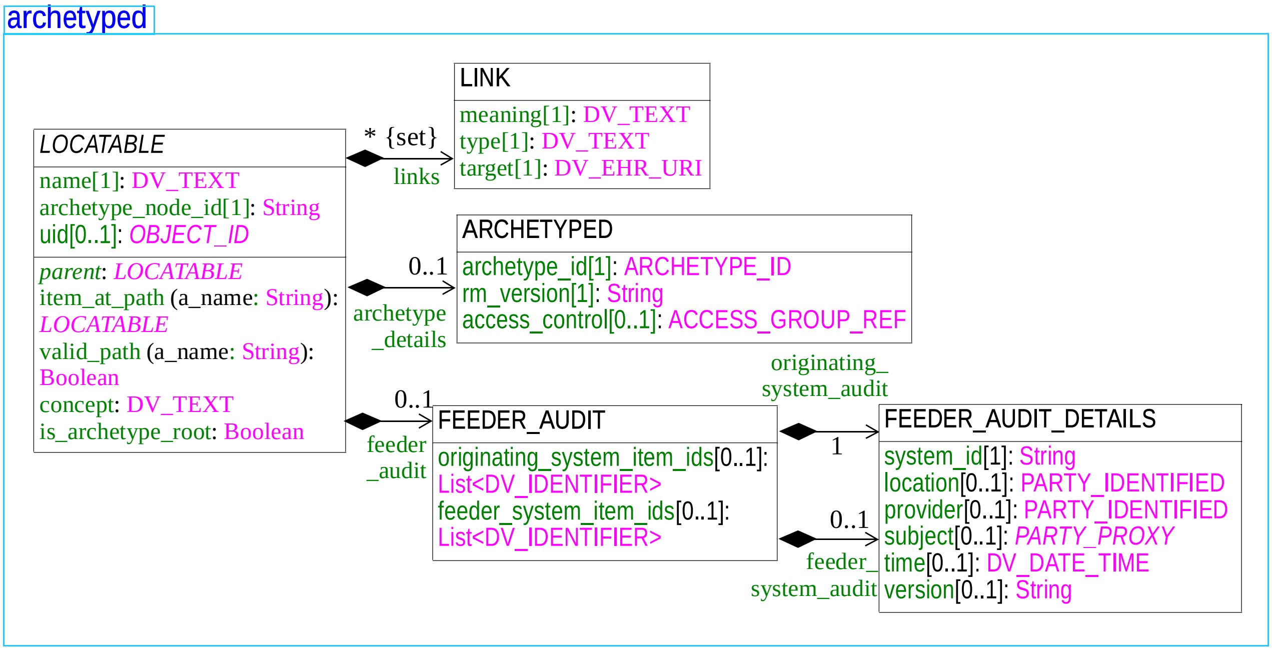

FIGURE 2 rm.common.archetyped Package

3.1.1 The Class LOCATABLE

Every structural class in the *open*EHR reference model inherits from the `LOCATABLE` class, ensuring it has both a runtime *name*, and an *archetype_node_id*. The *archetype_node_id* is the standardised semantic code for a node and comes from the corresponding node in the archetype used to create the data. The *name* attribute carries a name created at runtime. The “meaning” of any node is derived formally from the archetype by obtaining the “text” value for the *archetype_node_id* code from the archetype ontology, in the language required.

The *name* and *archetype_node_id* values are often the same semantically, but may differ. For example, in “problem/SOAP” Sections (i.e. headings), the name of a section at the problem level might be “diabetes”, but its meaning (obtained using the *archetype_node_id* value from the local ontology in its generating archetype) will be “problem”. The default value for *name* should be assumed to be the text value in the local language for the *archetype_node_id* code (extracted from the archetype during data construction) on the node in question, unless explicitly set otherwise. `LOCATABLE` also provides the attribute *archetype_details*, which is non-Void for archetype root points in data.

USE ARCHETYPE_ID AT ROOT POINTS

`LOCATABLE` objects may also have a *uid*, possibly implemented using an ISO Oid or a GUID. 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; root points might also 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.

The *parent* feature in `LOCATABLE` ensures that any `LOCATABLE` can reference its parent in the compositional hierarchy, and may be implemented in any way convenient.

3.1.2 Feeder System Audit

The data in any part of the EHR may be obtained from a *feeder* system, i.e. a source system which does not obey the versioning, auditing and content semantics of *open*EHR (data in the EHR which have been sourced from another *open*EHR system are dealt with in the Common IM, Change control section). The `FEEDER_AUDIT` class defines the semantics of an audit trail which is constructed to describe the origin of data which have been transformed into *open*EHR form and committed to the system. There are a number of important aspects to the problem of transforming data for committal into an *open*EHR system, dealt with in the following subsections.

Requirements

The model of Feeder audit is designed to satisfy the following requirements with respect to EHR content sourced from non-*open*EHR systems:

- record medico-legal audit information from the originating system (e.g. pathology lab system) similar to that captured in the `AUDIT_DETAILS` class in the `change_control` package;
- record information identifying the system from which the content was obtained (might not be the originating system);
- record sufficient information to distinguish incoming items from each other, and to enable the detection of duplicates and new versions of the same item.

Design Principles

The design of the Feeder audit part of the reference model is based on a generalised model of data communication in which various elements are identified, as follows:

the originating system: the computer system where the information item was initially created, e.g. the system at a pathology laboratory or a reporting system for a number of laboratories;

intermediate systems: any system which moves information from the originating system to *open*EHR system;

the feeder system: the intermediate system from which the information item was directly obtained by the *open*EHR system; this might be the originating system, or it may be a distinct intermediate system;

the committing openEHR system: the *open*EHR system where the information item is transformed into *open*EHR form and committed as a Composition;

openEHR converter: a component whose job it is to convert non-*open*EHR information into a form compliant with the *open*EHR reference model and chosen archetypes.

FIGURE 3 illustrates these elements, shown as a "feeder chain", along with typical meta-data available in messages from each system. In general, not much can be assumed about systems in the feeder chain. The originating system may or may not correspond to the place of the clinical activity - it is not uncommon for a pathology company to have a centralised report issuing location while having numerous physical laboratories. There is often limited consistency in the way identifiers are assigned, timestamps are created, and information is structured and coded. In general, information from a feeder system is in response to a request, often a pathology order, although the request/response pattern probably cannot be assumed in all cases.

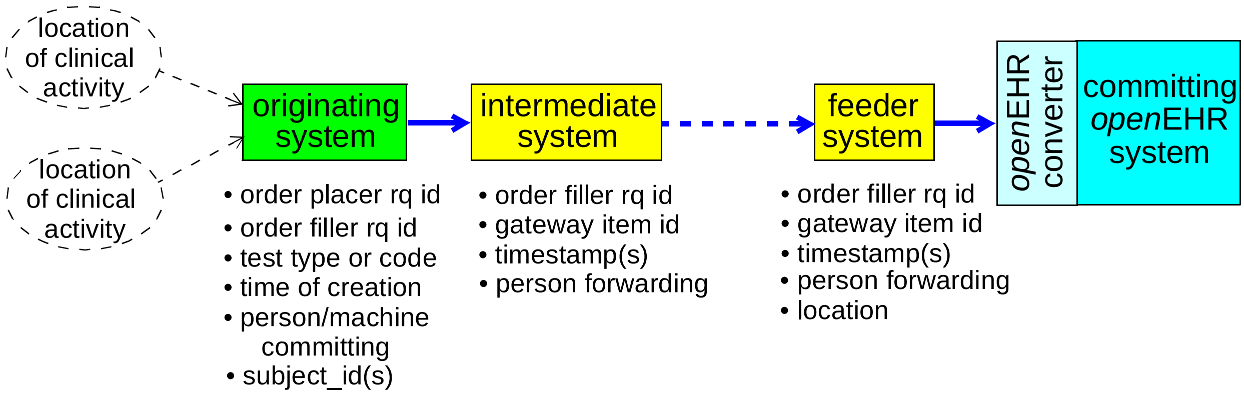

FIGURE 3 Abstract model of feeder chain

The idea underlying the *open*EHR Feeder audit model is that there are two groups of meta-data which should be recorded about an imported information item. The first is medico-legal meta-data about its creation: the system of origin, who created it and when it was created. The second is identifying meta-data for the item from the originating and feeder system, and potentially other intermediate systems in the feeder chain, where necessary to support duplicate detection, version detection and so on.

Meta-data

The potentially available medico-legal meta-data about the received item is as follows:

- identifier of the originating system (where the item was originally committed);
- identifier of the information item in the originating system;
- agent who committed the item;
- timestamp of committal or creation of the item;
- type of change, e.g. initial creation, correction (including deletion of a subpart), logical deletion (e.g. due to cancellation of order);
- status of information, e.g. interim, final;
- version id, where versioning is supported.

The above information is equivalent to the audit trail and versioning data captured when information created in an *open*EHR system is committed in a Composition version.

Various kinds of identifying information may be required including the following:

- subject identifier (often more than one, e.g. national patient id, GP’s local patient id, lab’s local patient id) are usually recorded and may be required for traceability purposes;
- subject identifier(s) may identify someone other than the subject of the record as being the subject of the incoming item;
- location of the feeder system;
- identifier of the feeder system (which may be one of many at the feeder system location);
- identifer the feeder system uses for the item in question (often known as an “accession id”);

- identifier of request or order to which the information is a response (sometimes known as a "placer's request id");
- identifier of the information item used by the originating system (sometimes known as a "filler's request id");
- timestamp(s) assigned by feeder system to the item.

Some or all of this information will usually be sufficient to perform a number of tasks as follows.

Traceability

The first is to support medico-legal investigation into the path of the information item through the health computing infrastructure. This requires the availability of sufficient identifier information that the origin of the information item can be traced.

Subject identifiers where available should be used to ensure that the received data go into the correct EHR, by ensuring that the relevant lookups in client directories or other lookup mechanisms can be effected. Again, in rare cases, the subject of the incoming data item may not necessarily be the subject the EHR - a test result may be made from a relative or other associate which will be stored in the patient's EHR.

Version Detection

The second is to detect new versions of an item (e.g. interim and final versions of a microbiology test result). This can usually be achieved by using various identifiers as well as the originating system version id and/or content status (interim, final etc). A new *open*EHR Composition version should always be created for each received version, even where the content does not change at all (e.g. a microbiology test where the result is "no growth" in both interim and final results).

Duplicate Detection

Another task is to disambiguate duplicates (often caused by failure of a network connection during sending) coming from the feeder system. In some cases however duplicates are erroneously given new ids by the feeder system, giving the receiver the impression of a new information. In such cases, a further item of meta-data may be required:

- hash or content signature generated (most likely by the converter) from the received information.

Differentially Coded Data

A further problem is that the originating system may send new versions of an item which are not complete in and of themselves, i.e. which only include new or changed elements with respect to a previous send of the same item. An example is a system which sends a correction to an HL7v2 blood test message, where the correction includes just the "serum sodium" data item. In this case, special processing will be required in the *open*EHR converter component, in order to regenerate a full data item from difference data when it is received. Such processing may also have to take account of deleted items.

In summary, the Feeder audit class design tries to accommodate the recording of as much of the above meta-data as is relevant in any particular case. It is up to the design of *open*EHR conversion front-end components as well as proper analysis of the situation to determine which identifiers are germane to the needs of traceability. In general, any meta-data of medico-legal significance should be captured where it is available.

Using Feeder Audit in Converted Data

Although the design of the *open*EHR converter is outside the scope of the current document, it is worth considering a common design approach, and where the FEEDER_AUDIT class fits in. An effective way of converting non-*open*EHR data such as HL7v2 messages, relational data etc, is in two steps. The first is to perform a 'syntactic' conversion to Compositions containing instances of the GENERIC_ENTRY class (described in the Integration IM), using 'legacy archetypes'. The resulting database will contain versioned Compositions containing GENERIC_ENTRY instances; logically this database does not contain EHRs but simply external data converted to *open*EHR computational form. The relevant FEEDER_AUDIT instances should be attached to the Compositions containing the corresponding GENERIC_ENTRY instances. The second step is to perform a 'semantic' conversion to subtypes of ENTRY, i.e. OBSERVATION, EVALUATION, INSTRUCTION and ACTION, according to standardised clinical archetypes. There are various possibilities for what to do with the Feeder audit. The minimum Feeder audit required on the final instance contains the originating system audit information, but none of the information to do with feeder or intermediate systems. This will satisfy medico-legal needs. Alternatively, a complete copy could be made, even though the feeder-related meta-data is probably only of use in the conversion environment. What the Feeder audit looks like in the EHR proper may depend on local legislation, norms or other factors. Completely alternative conversion processes are also possible, in which no intermediate form of data exists.

Structural Correspondence

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 relevant 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".

3.2 Class Descriptions

3.2.1 Class LOCATABLE

CLASS	*LOCATABLE (abstract)*	
Purpose	Root structural class of all information models. The *parent* feature may be implemented as a function or attribute.	
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.	
Abstract	**Signature**	**Meaning**
	parent: LOCATABLE	Parent of this node in compositional hierarchy.
Attributes	**Signature**	**Meaning**
0..1	**uid**: OBJECT_ID	Optional globally unique object identifier for root object of archetyped data structure.
1..1	**archetype_node_id**: String	Design-time archetype id of this node taken from its generating archetype; used to build archetype paths. Always in the form of an “at” code, e.g. “at0005”. This value enables a "standardised" name for this node to be generated, by referring to the generating archetype local ontology. At an archetype root point, the value of this attribute is always the stringified form of the *archetype_id* found in the *archetype_details* object.
1..1	**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.
0..1	**archetype_details**: ARCHETYPED	Details of archetyping used on this node.
0..1	**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.

CLASS	LOCATABLE (abstract)	
0..1	**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 'archetype_node_id' of the root node)
Invariant	***Archetype_node_id_valid***: archetype_node_id /= Void ***Name_exists***: name /= Void ***Links_valid***: links /= Void ***implies not*** links.empty ***Archetyped_validity***: is_archetype_root ***xor*** archetype_details = Void	

3.2.2 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`

CLASS	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 *archetype_node_id* 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**
1..1	**archetype_id**: `ARCHETYPE_ID`	Globally unique archetype identifier.
0..1	**access_control**: `ACCESS_GROUP_REF`	The access control settings of this component.
1..1	**rm_version**: `String`	Version of the *open*EHR reference model used to create this object. Expressed in terms of the release version string, e.g. "1.0", "1.2.4".
Invariant	***archetype_id_exists***: archetype_id /= Void ***rm_version_exists***: rm_version /= Void ***and then not*** rm_version.empty	

3.2.3 LINK Class

CLASS	LINK
Purpose	The `LINK` type defines a logical relationship between two items, such as two `ENTRY`s 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 `LINK`s 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. `ENTRY`s) can be expressed by using one, or more than one, respectively, `DV_LINK`s. Chains of links can be used to see "problem threads" or other logical groupings of items.

<table>
<tr><th>CLASS</th><th colspan="2">LINK</th></tr>
<tr><td>MisUse</td><td colspan="2">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 <code>ENTRY</code>s, <code>SECTION</code>s, <code>COMPOSITION</code>s and so on.</td></tr>
<tr><td>CEN</td><td colspan="2">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.</td></tr>
<tr><td>Synapses</td><td colspan="2">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.</td></tr>
<tr><td>GEHR</td><td colspan="2"><code>n/a</code></td></tr>
<tr><td>HL7v3</td><td colspan="2">The <code>ACT_RELATIONSHIP</code> class in some cases appears to correspond to <code>LINK</code>.</td></tr>
<tr><th>Attributes</th><th>Signature</th><th>Meaning</th></tr>
<tr><td>1..1</td><td>meaning: <code>DV_TEXT</code></td><td>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.

Values 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”.</td></tr>
<tr><td>1..1</td><td>type: <code>DV_TEXT</code></td><td>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.</td></tr>
<tr><td>1..1</td><td>target: <code>DV_EHR_URI</code></td><td>the logical “to” object in the link relation, as per the linguistic sense of the meaning attribute.</td></tr>
<tr><td>Invariant</td><td colspan="2">meaning_exists: meaning /= Void
type_exists: type /= Void
target_exists: target /= Void</td></tr>
</table>

3.2.4 FEEDER_AUDIT Class

CLASS	FEEDER_AUDIT	
Purpose	Audit and other meta-data for systems in the feeder chain.	
Attributes	**Signature**	**Meaning**
1..1	**originating_system_audit**: `FEEDER_AUDIT_DETAILS`	Any audit information for the information item from the originating system.
0..1	**originating_system_item_ids**: `List<DV_IDENTIFIER>`	Identifiers used for the item in the originating system, e.g. filler and placer ids.
0..1	**feeder_system_audit**: `FEEDER_AUDIT_DETAILS`	Any audit information for the information item from the feeder system, if different from the originating system.
0..1	**feeder_system_item_ids**: `List<DV_IDENTIFIER>`	Identifiers used for the item in the feeder system, where the feeder system is distinct from the originating system.
Invariants	***Originating_system_audit***: originating_system_audit /= Void	

3.2.5 FEEDER_AUDIT_DETAILS Class

CLASS	FEEDER_AUDIT_DETAILS	
Purpose	Audit details for any system in a feeder system chain. Audit details here means the general notion of who/where/when the information item to which the audit is attached was created. None of the attributes is defined as mandatory, however, in different scenarios, various combinations of attributes will usually be mandatory. This can be controlled by specifying feeder audit details in legacy archetypes.	
Attributes	**Signature**	**Meaning**
1	**system_id**: `String`	Identifier of the system which handled the information item.
0..1	**provider**: `PARTY_IDENTIFIED`	Optional provider(s) who created, committed, forwarded or otherwise handled the item.
0..1	**location**: `PARTY_IDENTIFIED`	Identifier of the particular site/facility within an organisation which handled the item. For computability, this identifier needs to be e.g. a PKI identifier which can be included in the *identifier* list of the `PARTY_IDENTIFIED` object.
0..1	**time**: `DV_DATE_TIME`	Time of handling the item. For an originating system, this will be time of creation, for an intermediate feeder system, this will be a time of accession or other time of handling, where available.
0..1	**subject**: *`PARTY_PROXY`*	Identifiers for subject of the received information item.
0..1	**version_id**: `String`	Any identifier used in the system such as “interim”, “final”, or numeric versions if available.
Invariants	***System_id_valid***: system_id /= Void ***and then not*** system_id.is_empty	

4 Generic Package

4.1 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.

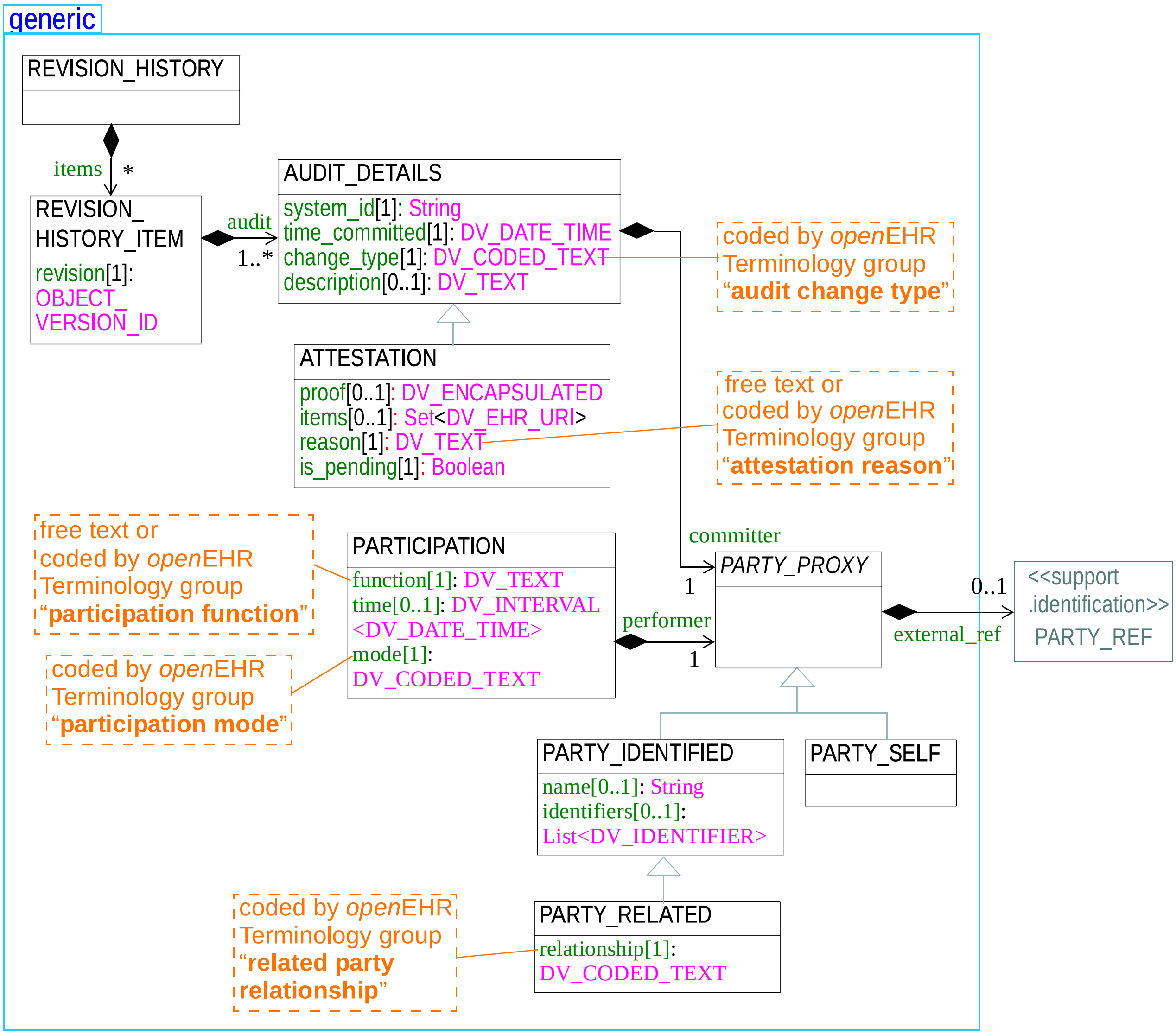

FIGURE 4 rm.common.generic Package

4.2 Design Principles

There are two ways to refer to an identity in the *open*EHR EHR: using PARTY_REF directly, which records an identifier of the party in some external system, and using PARTY_PROXY, which supports a small amount of descriptive data, depending on the subtype, and an optional PARTY_REF. The semantics of PARTY_REF are described in the Common IM, identification package, while the semantics of PARTY_PROXY and use of PARTY_REF in such entities are described below.

4.2.1 Referring to Demographic Entities

The approach taken in *open*EHR for representing demographic and user entities in the EHR data is based on the following assumptions:

- there is at least one human readable name or official identifier of the party, such as "Julius Marlowe, MD", "NHS provider number 1039385", or a system user id such as "Rahil.Azam";
- there might be data in a service external to the EHR for the party in question, such as a demographic, identity management or patient index service; if there is, we want to reference it;
- the subject of the record is never to be identified in any direct way (i.e. via the use of her name or other human-readable details), but may include a meaningless identifier in some external system.

The PARTY_PROXY class and subtypes model references to parties based on these assumptions. The semantics of PARTY_PROXY enable a flexible approach: in stricter environments that have identity management and demographic services, and where there is an entry in such a service for the party in question, PARTY_PROXY.*external_ref* will be non-Void, while in other environments, it will be empty.

The two subtypes correspond to the mutually distinct categories of the 'subject of the record', known as the 'self' party in *open*EHR, and any other party. Whenever the record subject has to be referred to in the record, an instance of PARTY_SELF is used, while PARTY_IDENTIFIED is used for all other situations. The latter class provides for optional human-readable *names* and formal *identifiers*, each keyed by purpose or meaning.

The RELATED_PARTY type is used whenever the relationship of the party to the record subject is required. Relationships are coded and include familial ones ('mother', 'uncle', etc) as well as relationships like 'donor', 'travelling companion' and so on.

PARTY_SELF and Referring to the Patient from the EHR

There are three schemes which are likely to be used for referring to patient (i.e. the record subject) demographic or patient master index (PMI) data from within the EHR, each likely to be valid in different circumstances. These are as follows.

- Once only in EHR.*subject* using the PARTY_SELF.*external_ref*. Since the EHR object is separate from the EHR contents, the root instance of PARTY_SELF will generally not be visible.
- Setting the *external_ref* in every instance of PARTY_SELF; this solution makes the patient external ref visible in every instance of PARTY_SELF, which is reasonable in a secure environment, and convenient for copying parts of the record around locally.
- On no instances of PARTY_SELF. This is the most secure approach, and means that the link between the EHR and the patient has to be done outside the EHR, by associating EHR.uid and the patient demographic/PMI identifier. This approach is more likely for more open environments.

All three schemes are supported by the *open*EHR model, and will probably all find use in different settings and EHR deployment types.

4.2.2 Participation

The Participation abstraction models *the interaction of some Party in 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_PROXY attribute in 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 via a PARTY_PROXY instsance, and records the function, time interval and (coded) mode of the participation. It can be used in any other reference model as required.

4.2.3 Audit Information

Audit Details

Three classes are provided to represent audit information. The first, AUDIT_DETAILS expresses the details that would be captured about a user when committing some information to a repository of some kind, which may be version controlled. It records committer, time, change type and description. Committer is recorded using a PARTY_PROXY, allowing for PARTY_SELF to be used when the committer is the record subject, and for other identifying information to be included for other users, expressed using PARTY_IDENTIFIED. The kind of identifying information used in PARTY_PROXY instances in AUDIT_DETAILS may be different from that used in COMPOSITION.*composer* or elsewhere, i.e. in the form of a system login identifier, e.g. "maxime.lavache@stpatricks.health.ie".

Revision History

The classes REVISION_HISTORY and REVISION_HISTORY_ITEM express the notion of a revision history, which consists of audit items, each associated with a revision number. An instance of the REVISION_HISTORY_ITEM class is designed to express the information that corresponds to an item in a revision history, i.e. a list of all audits relating to some information item. The revision is included to indicate which version/revision each audit corresponds to. These classes provide an interoperable definition of revision history for the VERSIONED_OBJECT and AUTHORED_RESOURCE classes.

4.2.4 Attestation

Attestation is another concept which occurs commonly in health information. An attestation is an explicit signing by one healthcare agent of particular content for various particular purposes, including:

- for authorisation of a controlled substance or procedure (e.g. sectioning of patient under mental health act);
- witnessing of content by senior clinical professional;
- indicating acknowledgement of content by intended recipient, e.g. GP who ordered a test result.

Here it is modelled as a subtype of AUDIT_DETAILS, meaning that it is logically a kind of audit, with additional information pertinant to the act of signing, namely 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 following items together:

- the identity of the attesting party;
- the thing attested to, e.g. a statement like "Do you agree that the form below is an accurate record 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 *reason* attribute is used to indicate why the attestation occurred, and is coded using the *open*EHR Terminology group "attestation reason", which includes values such as "authorisation" and "witnessed". The *is_pending* attribute marks the attestation as either having been done or awaiting completion depending on its value. This facilitates querying the record to find items needing to be signed or witnessed. When an attestation is required, the most common scenario will be that a Composition Version will be committed with an *audit* of type `ATTESTATION`, rather than just `AUDIT_DETAILS`; the *is_pending* flag will be set to True to indicate that the committed information needs to be signed by another person. When the latter happens, it will cause a new `ATTESTATION` object to be added to the `VERSION`.*attestations* list, this time with *is_pending* set to False, and the appropriate proof supplied. In general, a need for attestation followed by an attestation will be represented by two `ATTESTATION` objects, the first in the *is_pending* state, and the second with *is_pending* False, and containing the relevant proof.

4.3 Class Descriptions

4.3.1 PARTY_PROXY Class

CLASS	*PARTY_PROXY (abstract)*	
Purpose	Abstract concept of a proxy description of a party, including an optional link to data for this party in a demographic or other identity management system. Subtyped into `PARTY_IDENTIFIED` and `PARTY_SELF`.	
Attributes	**Signature**	**Meaning**
0..1	**external_ref**: `PARTY_REF`	Optional reference to more detailed demographic or identification information for this party, in an external system.
Invariant		

4.3.2 PARTY_SELF Class

CLASS	PARTY_SELF	
Purpose	Party proxy representing the subject of the record.	
Use	Used to indicate that the party is the owner of the record. May or may not have *external_ref* set.	
Inherit	`PARTY_PROXY`	
Attributes	**Signature**	**Meaning**

CLASS	PARTY_SELF
Invariant	

4.3.3 PARTY_IDENTIFIED Class

CLASS	PARTY_IDENTIFIED	
Purpose	Proxy data for an identified party other than the subject of the record, minimally consisting of human-readable identifier(s), such as name, formal (and possibly computable) identifiers such as NHS number, and an optional link to external data. There must be at least one of *name*, *identifier* or *external_ref* present.	
Use	Used to describe parties where only identifiers may be known, and there is no entry at all in the demographic system (or even no demographic system). Typically for health care providers, e.g. name and provider number of an institution.	
Misuse	Should not be used to include patient identifying information.	
Inherit	`PARTY_PROXY`	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**name**: `String`	Optional human-readable name (in String form).
0..1 (cond)	**identifiers**: `List<DV_IDENTIFIER>`	One or more formal identifiers (possibly computable).
Invariant	***Basic_validity*** name /= Void ***or*** identifiers /= Void ***or*** external_ref /= Void ***Name_validity***: name /= Void ***implies not*** name.is_empty ***Identifiers_validity***: identifiers /= Void ***implies not*** identifiers.is_empty	

4.3.4 PARTY_RELATED Class

CLASS	PARTY_RELATED	
Purpose	Denote a party and its relationship to the subject of the record.	
Use	Use where the relationship between the party and the subject of the record must be known.	
Inherit	`PARTY_IDENTIFIED`	
Attributes	**Signature**	**Meaning**
1..1	**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”.

CLASS	PARTY_RELATED
Invariants	***Relationship_valid***: relationship /= Void ***and then*** terminology(“openehr”).codes_for_group_name(“related party relationship”, “en”) .has(relationship.defining_code)

4.3.5 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.	
HL7v3	RIM Participation class.	
Attributes	**Signature**	**Meaning**
1..1	**performer**: `PARTY_PROXY`	The id and possibly demographic system link of the party participating in the activity.
1..1	**function**: `DV_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 cannot be limited to the HL7v3:ParticipationFunction vocabulary, since it is too limited and hospital-oriented.
1..1	**mode**: `DV_CODED_TEXT`	The mode of the performer / activity interaction, e.g. present, by telephone, by email etc.
0..1	**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.
Invariant	***Performer_exists***: performer /= Void ***Function_valid***: function /= Void ***and then*** function.generating_type.is_equal(“DV_CODED_TEXT”) ***implies*** terminology(“openehr”).codes_for_group_name(“participation function”, “en”) .has(function.defining_code) ***Mode_valid***: mode /= Void ***and*** terminology(“openehr”).codes_for_group_name(“participation mode”, “en”).has(mode.defining_code)	

4.3.6 ATTESTATION Class

CLASS	ATTESTATION	
Purpose	Record an attestation of a party (the committer) to item(s) of record content. The type of attestation is	
Inherit	AUDIT_DETAILS	
Attributes	**Signature**	**Meaning**
0..1	**proof**: DV_ENCAPSULATED	Proof of attestation.
0..1	**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.
1..1	**reason**: DV_TEXT	Reason of this attestation. Optionally coded by the *open*EHR Terminology group "attestation reason"; includes values like "authorisation", "witness" etc.
1..1	**is_pending**: Boolean	True if this attestation is outstanding; False means it has been completed.
Invariants	***Items_validity***: items /= Void ***implies not*** items.is_empty ***Reason_validity***: reason /= Void ***and then*** (reason.generating_type.is_equal("DV_CODED_TEXT") ***implies*** terminology("openehr").codes_for_group_name("attestation reason", "en").has(reason.defining_code))	

4.3.7 AUDIT_DETAILS Class

CLASS	AUDIT_DETAILS	
Purpose	The set of attributes required to document the committal of an information item to a repository.	
Synapses	Composition class	
GEHR	G1_COMMIT_AUDIT	
Attributes	**Signature**	**Meaning**
1..1	**system_id**: String	Identity of the system where the change was committed.

CLASS	AUDIT_DETAILS	
1..1	**committer**: `PARTY_PROXY`	Identity and optional reference into identity management service, of user who committed the item.
1..1	**time_committed**: `DV_DATE_TIME`	Time of committal of the item.
1..1	**change_type**: `DV_CODED_TEXT`	Type of change. Coded using the *open*EHR Terminology “audit change type” group.
0..1	**description**: `DV_TEXT`	Reason for committal.
Invariants	***System_id_exists***: system_id /= Void ***and then not*** system_id.empty ***Committer_exists***: committer /= Void ***Time_committed_exists***: time_committed /= Void ***Change_type_exists***: change_type /= Void ***and then*** terminology(“openehr”).codes_for_group_name(“audit change type”, “en”).has(change_type.defining_code)	

4.3.8 REVISION_HISTORY Class

CLASS	REVISION_HISTORY_ITEM	
Purpose	Defines the notion of a revision history of audit items, each associated with the revision at which that change occurred.	
Attributes	Signature	Meaning
1..1	**items**: List<REVISION_HISTORY_ITEM>	The items in this history.
Invariants	***Items_exists***: items /= Void	

4.3.9 REVISION_HISTORY_ITEM Class

CLASS	REVISION_HISTORY_ITEM	
Purpose	An entry in a revision history, corresponding to a version from a versioned container. Consists of `AUDIT_DETAILS` instances with revision identifier of the revision to which the `AUDIT_DETAILS` intance belongs.	
Attributes	**Signature**	**Meaning**
1..*	**audit**: `AUDIT_DETAILS`	The audit information for this revision.
1..1	**revision**: `String`	Revision identifier of this audit trail.
Invariants	***Audit_exists***: audit /= Void ***Revision_exists***: revision /= Void ***and then not*** revision.empty	

5 Directory Package

5.1 Overview

The `directory` package is illustrated in FIGURE 5. It provides a simple abstraction of a versioned folder structure. The `VERSIONED_FOLDER` class is the binding of `VERSIONED_OBJECT<T>` to `FOLDER`, i.e. it is a `VERSIONED_OBJECT<FOLDER>`. This means that each of its versions is a `FOLDER` structure. It provides a means of versioning `FOLDER` structures over time, which is useful in the EHR, Demographics service or anywhere else where Folders are used to group things. A `FOLDER` instance is simple: it contains more `FOLDER`s and/or items, which are references to other (usually versioned) objects. A `FOLDER` structure is therefore like a directory containing references to objects. Since they are only references, multiple references to the same object are possible, allowing the structure to be used to mutiply classify other objects. If it is used with `VERSIONED_COMPOSITION`s for example, the folders might be used to represent episodes and at the same time problem groups.

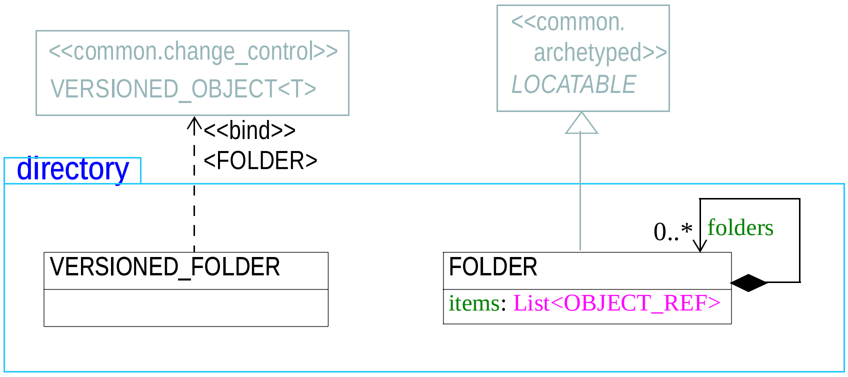

FIGURE 5 common.directory Package

`FOLDER` structures inside the `VERSIONED_FOLDER` are archetypable structures, and `FOLDER` archetypes can be created in the same fashion as say `SECTION` archetypes for the EHR.

5.1.1 Paths

Directory paths are built using the *name* attribute values inherited from `LOCATABLE` into each `FOLDER` object. In real data, these will usually be derived from the value of the *archetype_node_id* attribute, plus a uniqueness modifier if required. Example paths:

```
/folders[hospital episodes]/items[]
/folders[patient entered data]/folders[diabetes monitoring]
/folders[homeopathy contacts]
```

Uniqueness modifiers are appended in brackets, and are only needed to differentiate folders at the same node that would otherwise have the same names, e.g.

```
[hospital episodes]
[hospital episodes(car accident Aug 1998)]
```

5.2 Class Descriptions

5.2.1 VERSIONED_FOLDER Class

CLASS	VERSIONED_FOLDER	
Purpose	A version-controlled hierarchy of FOLDERs giving the effect of a directory.	
Inherit	VERSIONED_OBJECT <FOLDER>	
Attributes	Signature	Meaning
Invariants		

5.2.2 FOLDER Class

CLASS	FOLDER	
Purpose	The concept of a named folder.	
CEN	FOLDER class	
Synapses	RecordFolder class	
Inherit	LOCATABLE	
Attributes	Signature	Meaning
0..1	**folders**: List<FOLDER>	Sub-folders of this FOLDER.
0..1	**items**: List<OBJECT_REF>	The list of references to other (usually) versioned objects logically in this folder.
Invariants	***Folders_valid***: folders /= Void ***implies not*** folders.empty	

6 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 change more than one versioned object in the repository, and the set of such changes - a "change-set" - constitutes the logical unit of change to the repository, taking it from one valid state to the next.

These concepts are well-known in configuration management (CM), and are used as the basis for most software and other change management systems, including numerous free and commercial products available 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 6.

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;

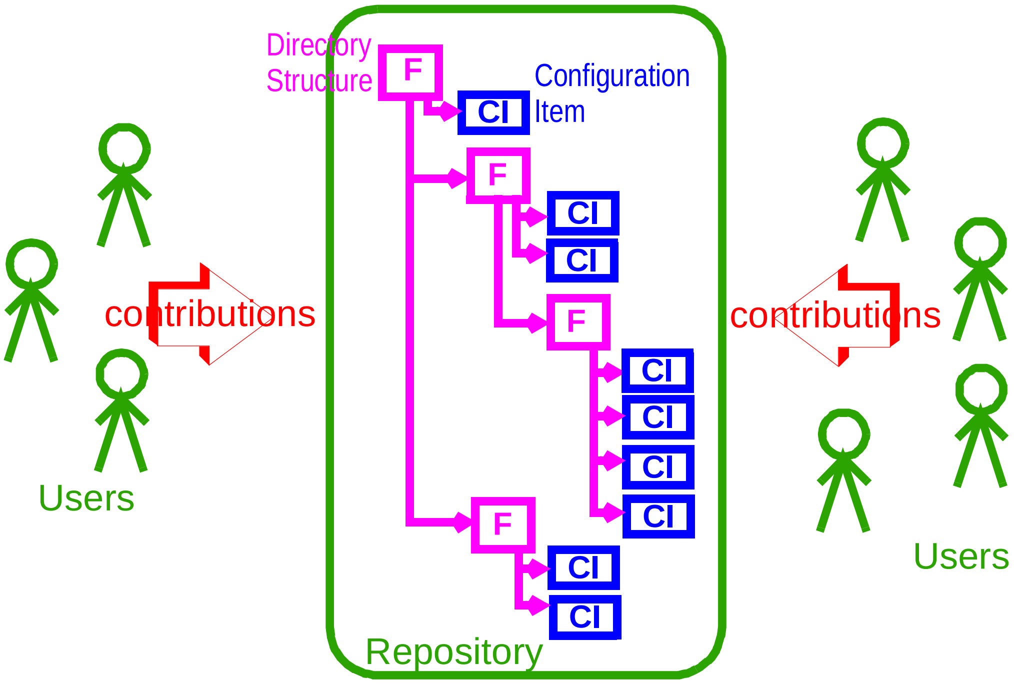

FIGURE 6 General Structure of a Controlled Repository

- 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 7 shows a the effect of four Contributions (indicated by blue ovals on the left hand side) to a repository containing a number of CIs (that the directory tree is not shown for the sake of simplicity). 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 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).

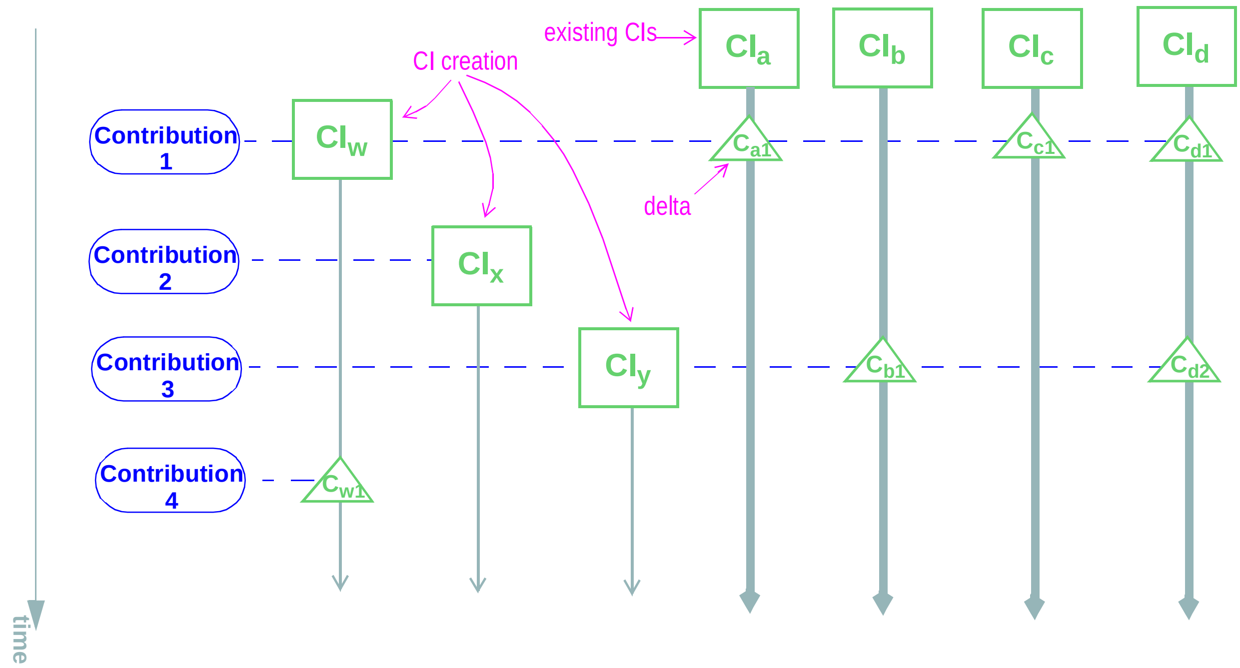

FIGURE 7 Contributions to the Repository (delta form)

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

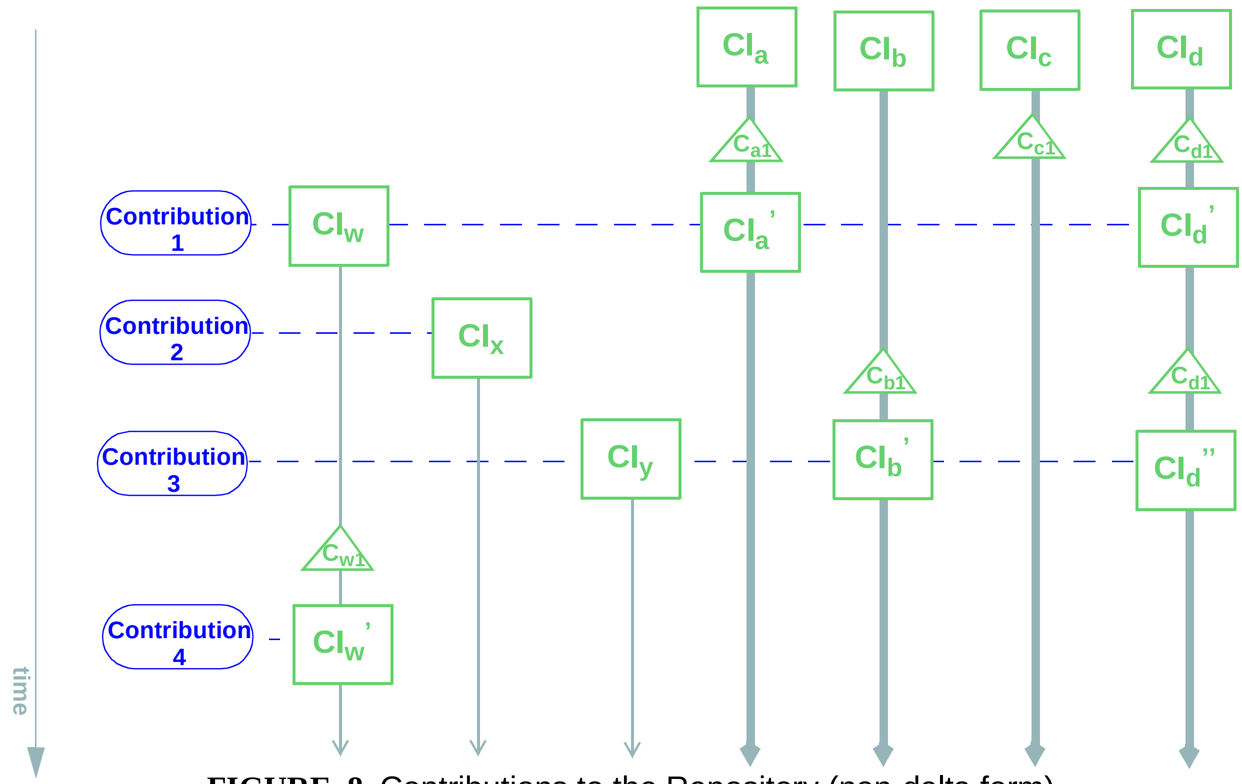

FIGURE 8 Contributions to the Repository (non-delta form)

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 `VERSIONED_OBJECT<T>`;
- `CONTRIBUTION`s;
- 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 therefore provides a guide to the definition of actual controlled repositories, such as an EHR, rather than a formal specification for them.

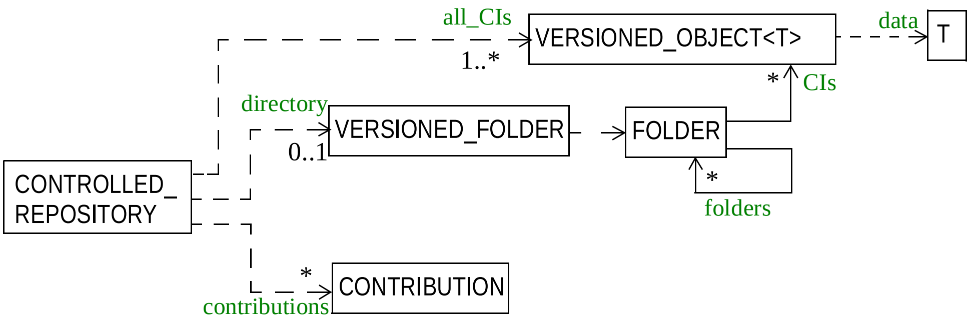

FIGURE 9 Informal Model of Change-controlled Repository

6.3 Formal Model

6.3.1 Overview

FIGURE 10 illustrates a formal model of a version repository. In this model, the class `VERSIONED_OBJECT<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, audit trails, and any attestations applied to the version. Both `VERSIONED_OBJECT<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 `VERSIONED_OBJECT` are of the same type, such as `COMPOSITION`, `FOLDER`, or `PARTY` and that the repository itself is properly typed.

Each `VERSIONED_OBJECT` has a unique identifier recorded in the *uid* attribute (an `OBJECT_ID` typically containing a GUID), and a reference to the owning object (e.g. the owning EHR) in the *owner_id* attribute (this is typically also a GUID). The latter helps ensure that in storage systems, versioned objects are always correctly allocated to their enclosing repository, such as an EHR.

The `VERSION<T>` defines the data and meta-data of a single version in a `VERSIONED_OBJECT`. The attributes *owner_id*, *system_id_chain*, and *version_tree_id* together constitute a globally unique version identifier, enabling Versions to be identified and merged correctly regardless of copying and sub-

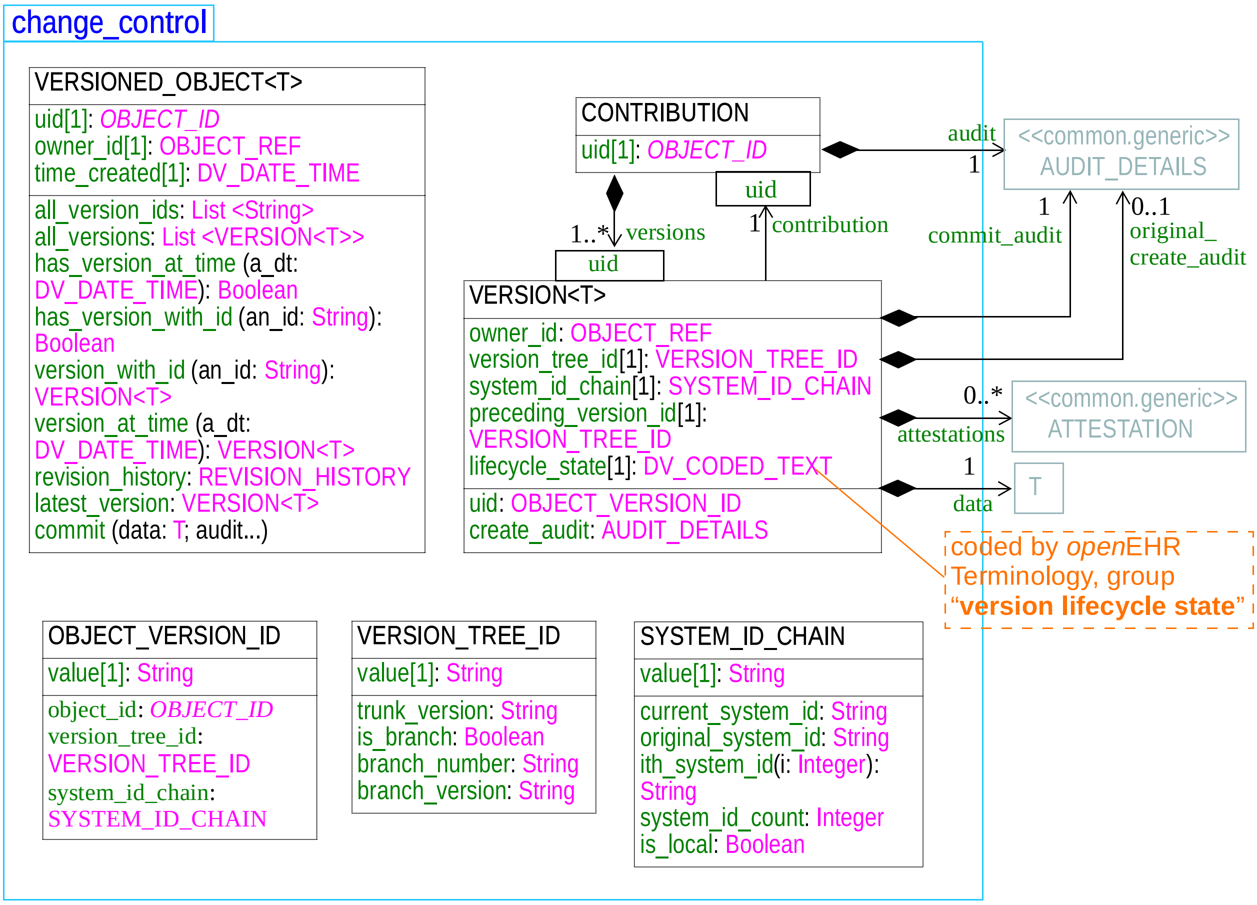

FIGURE 10 rm.common.change_control Package

sequent changes. The *uid* function returns these three identifiers in a single OBJECT_VERSION_ID instance. The details of the version identification scheme are described below.

6.3.2 Audits

There are two audit attributes in VERSION<T>: *commit_audit* and *original_create_audit*, both of type AUDIT_DETAILS. This class defines a set of attributes which form an audit trail, namely *system_id*, *committer*, *time_committed*, *change_type*, and *description*. Two audit trails are possible, to support the faithful retention of audit information even when versions are copied elsewhere. The scheme is as follows. Whenever a new version of content is committed to a VERSIONED_OBJECT, a *commit_audit* is created. If the content committed was originally created somewhere else, and is now being committed as a first-generation copy to a VERSIONED_OBJECT in another system, the original *commit_audit* is copied to the *original_create_audit* attribute. If copying of the version happens more than once, the *original_create_audit*, once set, is not subsequently changed: it always contains the audit corresponding to the original creation while the audits of intermediate commits of the copy are not retained. The function *create_audit* always returns a value; if the *commit_audit* is also the audit corresponding to creation it is returned, else the value of *original_create_audit* is returned.

The CONTRIBUTION class also contains an *audit* attribute. Whenever a CONTRIBUTION is committed, this attribute captures to the time, place and committer of the committal act; these three attributes (*system_id*, *committer*, *time_committed* of AUDIT_DETAILS) are copied into the corresponding attributes of the *commit_audit* of each VERSION included in the CONTRIBUTION. This is done to enable sharing of versioned entities independently of which Contributions they were part of.

6.3.3 Version Lifecycle

Versioned content has a lifecycle state associated with it, modelled using the VERSION.*lifecycle_state* attribute, which is coded from the *open*EHR Terminology “version lifecycle state” group. The possible values include “incomplete”, “awaiting_approval”, “complete” and “deleted”, each corresponding to a specific state of the content. Generally content will be committed in the “finished” state. However, in some circumstances, e.g. because the author has run out of time to finish writing part of the Composition, or due to an emergency, it may be committed as “unfinished” meaning that it is incomplete, or at least, unreviewed. In hospitals this is a common occurrence. Unfinished 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.

Compositions which need approval (e.g. to be sighted and signed by a more senior staff member) are saved in the “awaiting_attestation” state.

Going from “unfinished” to “finished” almost always corresponds to a change in content, and corresponds to a new VERSION regardless. This modelling approach allows such content to exist on the EHR system, but to be flagged as unfinished when viewed by a user.

The *lifecycle_state* values are given by the “version lifecycle state” group in the *open*EHR Terminology.

6.3.4 Attestation

The *attestations* attribute allows attestations to be associated with the data in the version. Attestations are treated in *open*EHR as a kind of audit, with additional attributes, and are described in detail in the Common IM. They 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. Normally, attestations refer to the entire version to which they are attached. However, it is possible for an ATTESTATION instance to refer to some finer-grained item within the data of the version, such as a single ENTRY within a COMPOSITION. If in subsequent versions, such an item is not changed (e.g. a different ENTRY in the same COMPOSITION is altered), then the relevant VERSION instances also need to refer to the ATTESTATION instances which remain valid. Since ATTESTATIONs are considered immutable objects once created, it does not really matter whether this is done by referring to a shared ATTESTATION instance, or by the use of copies.

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.

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 VERSIONED_OBJECT<COMPOSITION> is defined.

6.3.5 Semantics of Copying in Distributed Systems

In *open*EHR, the only unit of copying that satisfies traceability requirements and can be used among distributed *open*EHR servers is the VERSION. In order to copy a COMPOSITION or even an OBSERVATION somewhere else, its enclosing VERSION<T> object must be sent. In the case of the COMPOSITION type, a VERSION<COMPOSITION> object is sent. At the receiving system two things can happen.

If it is the first time any version of the item logically identified by its VERSIONED_OBJECT.*uid* was received from the originating system, a new VERSIONED_OBJECT<T> (e.g. VERSIONED_OBJECT<COMPOSITION>) is created, with its *uid* set to the same value as the received VERSION.*owner_id* (i.e. the same as the relevant VERSIONED_OBJECT.*uid* in the originating system). This establishes the newly created VERSIONED_OBJECT as being the same as the one from which it was copied. The received VERSION is then committed in the normal way (i.e. as part of a Contribution), with its *version_tree_id* and *system_id_chain* attributes carrying their original values. Thus, if a VERSION was received with *version_tree_id* = “2” and *system_id_chain* = “au.gov.health.rdh.ehr3”, these details will be retained in the target system (whose system id might be “au.gov.health.rbh.svrB”).

The alternative situation is if some version of the original item had already been copied; in this case, the newly received VERSION will be committed to the VERSIONED_OBJECT that must already have been created, with its *version_tree_id* and *system_id_chain* attributes intact.

In both cases, the *commit_audit* of the VERSION object reflects the act of local committal, while the *original_create_audit* will contain the audit from the committal that occurred when the object was first created.

In many cases, the received information will remained as is for the duration. However, in some cases, users at the receiver system might want to make modifications as well. This is likely to happen in the case of information items representing things like medication lists and allergies. When new versions are added to a copied object, branching numbering is used in the *version_tree_id*, while the local system id is appended to a copy of the previous *system_id_chain* attribute. The consequences of this are described in detail in the following section on version identification.

6.3.6 Version Identification

The following versioning scheme is adapted from the work of Hnìtynka and Plášil [3].

Local Versioning

The *version_tree_id* attribute of VERSION identifies a version of an item with respect to other versions in the same tree. The requirements here are the same as for typical versioning systems in use in software configuration management, and are as follows:

- to encode the relationship between versions in the version id, that is to say, version ids are constructed that given a series of ids, the relative positions in the tree can be determined;
- to distinguish error corrections from content changes;
- to allow for branches, so that variants of a particular node can be created; this might be done due to translation, or for teaching purposes.

A suitable scheme satisfying the above requirements for health information is the simplest possible: a single number representing the version. Versioning starts at 1 and continues by single increments. The succession of version identifiers formed by changes over time is known as the “trunk” of the version tree.

To support branching, a further pair of numbers is added. The first number identifies the branch (e.g. the 1st branch, 2nd branch etc from that trunk node), while the second identifies the version; this latter starts at 1, in the same way as the trunk version number. The result of this is that version numbers like 1.1.1 (first version of first branch from trunk node 1), 2.3.3 (3rd version of 3rd branch from trunk node 2) become possible. Inside *open*EHR systems where sharing with other systems does not occur, it is expected that branched versioning will be used extremely rarely; translation is likely to be the only reason (for example if a portuguese translation of an english language version is made).

Distributed Versioning

However, in a distributed environment where copying and subsequent modification can be made, there are more requirements of a versioning scheme, as follows:

- it must be possible for an item to be copied and for local modifications then to be made;
- it must be possible to send later versions from a source system to a target system that has already received earlier versions, and for these versions not to clash;
- it must be guaranteed that any version of any object is uniquely identified globally, no matter whether it is a locally created trunk version, a locally created branch version or a change made to a copied version.

To satisfy these needs, an addition is made to the identification scheme in the form of an attribute called *system_id_chain* that represents the identifiers of systems involved in creating and modifying the data. Whenever a new version of a particular Versoined object (with a particular uid) is created locally, *system_id_chain* is set to the identifier of the local system; the result is that versions are now identified as the tuple of the *uid, system_id_chain* and the *version_tree_id* attribute. This situation is illustrated on the left hand-side of FIGURE 11 (first shown version is *uid*=1; *system_id_chain*=“sysA”; *version_tree_id*=“1”).

However, this is not sufficient to cope with the scenario of copying (copy #1 in the figure) followed by subsequent versioning in the target system. In this situation, two things are needed to ensure that such versions are indeed globally unique. The first is that branched version identifiers are used; this ensures that the modifications now being made in the target system are considered in a global sense as logical branches or variants rather than trunk versions which are made in the originating system. It also allows later trunk versions from the originating system to be copied at some future time to the target system (copy #2 & #3 in the figure) with out version number clashes. The second is that the target system id (i.e. the identifier of the system where the modification is now being made to the copied version) is prepended to the *system_id_chain* attribute; this ensures that clahes don’t occur between branches made in the originating system and those made in the target system do not clash; in the figure, *system_id_chain* being set to “sysB::sysA” in the first grey version (1.1.1) in sysB prevents this version from being confused with the sysA/1.1.1 version copied from sysA. For this reason, the new attribute is called *system_id_chain* rather than just *system_id*.

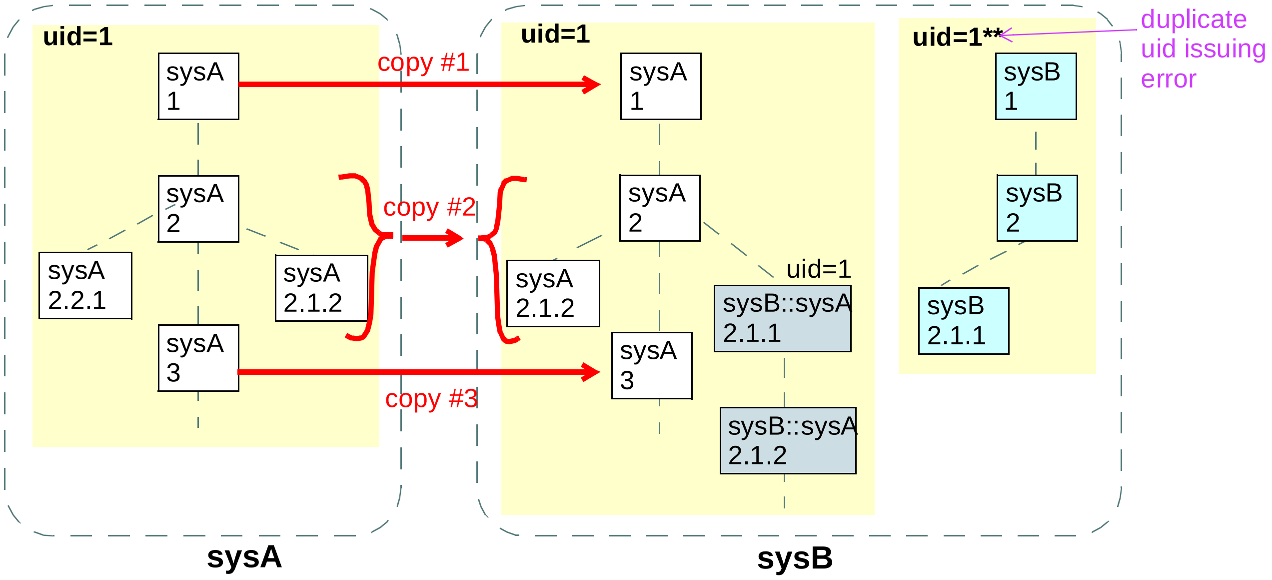

FIGURE 11 Versioning in a distributed environment

System id name chaining also acts as a protection against the (supposedly) unlikely occurrence of the same uid being issued in two systems, shown in the figure with the light blue version tree on the right hand side (*uid*=1**; *system_chain_id*=sysB; *version_tree_id*=1). If GUIDs are used, there is no guarantee that this will not happen, although it is statistically unlikely; it could also happen with Oids, due to incorrect issuing.

The format of the *system_id_chain* attribute is a double colon ("::") separated series of system identifiers, which themselves are not officially defined within *open*EHR. However, the most practical kind of identifier is probably a reverse internet node identifier, as recommended in [3]. This is because it is easy for the EHR or other systems known to use their own identifier without reference to an outside service (essentially, system administrators and the DNS have already performed this work). Disconnected desktop systems, such as private GP and patient computers will not normally have a domain or reliable host identifier. However, this shortcoming could be used as a reason for establishing reliable domain-based identifiers, e.g. within local health networks, which are now common in most companies. Note that it does not matter what the domain identifier is, as long as it is globally unique.

One rule is required to make this system work, as follows: when copies of versions are made to another system, branch versions from the originating system cannot be copied without their corresponding trunk versions.

6.3.7 Semantics of Moving VERSIONED_OBJECTs

It will not be uncommon that whole `VERSIONED_OBJECTS` need to be moved to another system, e.g. due to a move of a complete patient record (due to the patient moving), or even the relocation of a complete EHR data centre. The semantics of a move are different from those of copying: with a move, there is no longre a source instance, and the destination instance becomes the primary instance. When the move is effected, the identifier of the system in which the `VERSIONED_OBJECT` now exists will usually be different from what it was before. As a consequence, subsequent versions of the content created in a moved version container will now have the *system_id_chain* set to the id of new system only (rather than the concatenation of the new and old system ids, as happens in the copying case). This creates another variation on the version lineage, one in which the *system_id_chain* value can change in the trunk line, as shown in FIGURE 12.

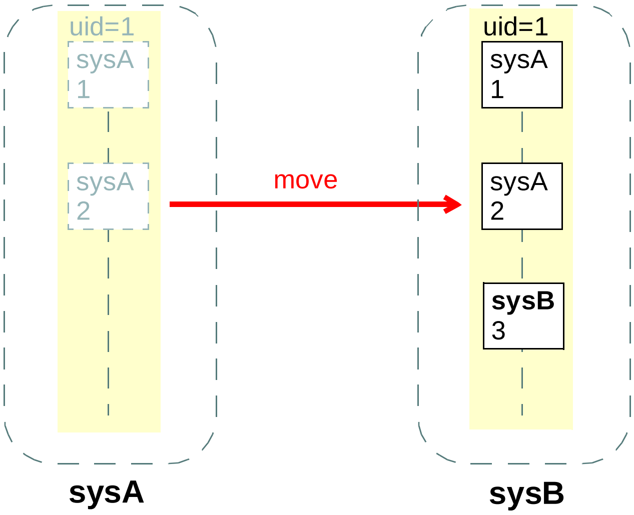

FIGURE 12 A Moved Version Container

6.3.8 Transaction Semantics of Contributions

In terms of database management, Contributions are similar to nested transactions. An 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.

6.4 Class Descriptions

6.4.1 VERSIONED_OBJECT Class

CLASS	VERSIONED_OBJECT<T>	
Purpose	Version control abstraction, defining semantics for versioning one complex object.	
Attributes	**Signature**	**Meaning**
1..1	**uid**: `OBJECT_ID`	Unique identifier of this version repository.
1..1	**owner_id**: `OBJECT_REF`	Reference to object to which this versioned repository belongs, e.g. the id of the containing EHR.
1..1	**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.

CLASS	VERSIONED_OBJECT<T>	
	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'.
	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.
	revision_history: `REVISION_HISTORY`	History of all audits and attestations in this versioned repository.
	commit (an_audit: `AUDIT_DETAILS`; 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 VERSION Class

CLASS	VERSION<T>
Purpose	Versionable objects, with create and commit audit trails containing details of committer, time and place.

CLASS	VERSION<T>	
Attributes	**Signature**	**Meaning**
1..1	**data**: `T`	The data being versioned.
0..1	**attestations**: `List <ATTESTATION>`	Set of attestations relating this version.
1..1	**commit_audit**: `AUDIT_DETAILS`	Audit trail corresponding to the committal of this version to the `VERSION_REPOSITORY` where is it currently located.
0..1 (cond)	**original_create_audit**: `AUDIT_DETAILS`	Audit trail corresponding to the creation and first-time committal of this version to the `VERSION_REPOSITORY` where it was first located. If Void, the *commit_audit* also represents the act of creation of the content. This attribute is mandatory when the version has been copied.
1..1	**version_tree_id**: `VERSION_TREE_ID`	Tree identifier of this version with respect to other versions in the same `VERSIONED_OBJECT`, as either 2 or 4 dot-separated numbers, e.g. 1.1, 2.1.1.4.
1..1	**system_id_chain**: `SYSTEM_ID_CHAIN`	Chain of identifiers of systems that created and modified the content of this version, starting at most recent modifier.
1..1	**preceding_version_id**: `VERSION_TREE_ID`	Tree identifier of the version on which this version was based.
1..1	**owner_id**: `OBJECT_REF`	A copy of the uid of the `VERSIONED_OBJECT` to which this version was added.
1..1	**contribution**: `OBJECT_REF`	Contribution in which this version was added.
1..1	**lifecycle_state**: `DV_CODED_TEXT`	Lifecycle state of the content item in this version.
Functions	**Signature**	**Meaning**
	create_audit: `AUDIT_DETAILS` ***ensure*** (is_local **and** Result = commit_audit) or (not is_local **and** Result = original_create_audit)	Audit trail corresponding to the committal of this version when the content was created. If it was created locally, then the result is the value of *commit_audit*, else it is the value of *original_create_audit*.
	is_branch: `Boolean`	True if this Version represents a branch.

CLASS	VERSION<T>	
	is_local: `Boolean`	True if this Version was created for the first time in the current system.
	uid: `OBJECT_VERSION_ID`	Unique identifier of this version, derived from *owner_id*, *version_tree_id* and *system_id_chain*.
Invariant	***version_tree_id_valid***: version_tree_id /= Void ***system_id_chain_valid***: system_id_chain /= Void ***preceding_version_id_valid***: preceding_version_id /= Void ***owner_id_exists***: owner_id /= Void ***lifecycle_state_valid***: lifecycle_state /= Void **and then** terminology("openehr").codes_for_group_name("version lifecycle state", "en").has(lifecycle_state.defining_code) ***commit_audit_exists***: commit_audit /= Void ***original_create_audit_validity***: original_create_audit /= Void **xor** is_local ***attestations_valid***: attestations /= Void **implies not** attestations.is_empty ***Contribution_exists***: contribution /= Void ***uid_valid***: uid /= Void **and** uid.version_tree_id.value(version_tree_id.value) and uid.system_id_chain.value.is_equal(system_id_chain.value) and uid.object_id.value.is_equal(owner_id.id.value) ***data_valid***: data /= Void ***Is_local_validity***: is_local = uid.is_local	

6.4.3 CONTRIBUTION Class

CLASS	CONTRIBUTION	
Purpose	Documents a contribution of one or more versions added to a change-controlled repository.	
Attributes	**Signature**	**Meaning**
1..1	**uid**: `OBJECT_ID`	Unique identifier for this contribution.
1..1	**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`.
1..1	**audit**: `AUDIT_DETAILS`	Audit trail corresponding to the committal of this Contribution.
Invariants	***uid_exists***: uid /= Void ***audit_exists***: audit /= Void ***Versions_valid***: versions /= Void ***and then not*** versions.empty ***Description_exists***: audit.description /= Void	

6.4.4 OBJECT_VERSION_ID Class

CLASS	OBJECT_VERSION_ID	
Purpose	Globally unique identifier for one version of a versioned object.	
Attributes	**Signature**	**Meaning**
1..1	**value**: `String`	String form of this identifier. Format is: {<object_id>;<version_tree_id>;<system_id_chain>}.
Functions	**Signature**	**Meaning**
	object_id: `OBJECT_ID`	Unique identifier for logical object of which this identifier identifies one version; normally the *object_id* will be the unique identifier of the version container containing the version referred to by this `OBJECT_VERSION_ID` instance.
	version_tree_id: `VERSION_TREE_ID`	Identifier for the version in the version tree.
	system_id_chain: `SYSTEM_ID_CHAIN`	Chain of system identifiers denoting (starting from the first) the most recent modifying system to the original creating system for the content carried in this version.
Invariants	***Value_valid***: value /= Void ***and then not*** value.is_empty ***Object_valid***: object_id /= Void ***Version_tree_id***: version_tree_id /= Void ***System_id_chain***: system_id_chain /= Void	

6.4.5 VERSION_TREE_ID Class

CLASS	VERSION_TREE_ID	
Purpose	Version tree identifier for one version.	
Attributes	**Signature**	**Meaning**
1..1	**value**: `String`	String form of this identifier. Format is: <trunk_version>[.<branch_number>.<branch_version>].
Functions	**Signature**	**Meaning**
	trunk_version: `String`	Trunk version number.

CLASS	VERSION_TREE_ID	
	branch_number: `String`	Number of branch from the trunk point <major_version>.<minor_version>
	branch_version: `String`	Version of the branch.
	is_branch: `Boolean`	True if this version identifier represents a branch, i.e. has branch_number and branch_version parts.
Invariants	***Value_valid***: value /= Void **and then not** value.is_empty ***Trunk_version_valid***: trunk_version /= Void **and then** trunk_version.is_integer ***Branch_number_valid***: branch_number /= Void **and then** branch_number.is_integer ***Branch_version_valid***: branch_version /= Void **and then** branch_version.is_integer ***Branch_validity***: (branch_number = Void **and** branch_version = Void) **xor** (branch_number /= Void **and** branch_version /= Void) ***Is_branch_validity***: is_branch **xor** branch_version = Void	

6.4.6 SYSTEM_ID_CHAIN Class

CLASS	SYSTEM_ID_CHAIN	
Purpose	Chain of system ids used to indicate lineage of a version with respect to location. Identifiers are not repeated. A value of “sysB::sysA” means that the item was created in a previous version in sysA, was copied to sysB, where the current version was made.	
Attributes	**Signature**	**Meaning**
1..1	**value**: `String`	String form of this identifier. Format is: sys_id{[::sys_id]}, i.e. reverse internet host addresses separated by the string “::” (double colon), starting with the current system id and ending with the original system id (i.e. newest to oldest). Suggested form for each system id is reverse domain of host where information created.
Functions	**Signature**	**Meaning**
	current_system_id: `String`	Identifier of the current system where the current version was created; returns the first system in the chain list.
	original_system_id: `String`	Identifier of the system where the content of the current version was originally created. Returns the last system in the chain list.
	ith_system_id(i: `Integer`): `String` ***require*** i > 0 and i <= system_id_count ***ensure*** Result /= Void **and then not** Result.is_empty	The i’th system_id in the chain.
	system_id_count: `Integer`	Number of ids in the chain.
	is_local: `Boolean`	True if the chain contains only one system id, being the id of the current system.

CLASS	SYSTEM_ID_CHAIN
Invariants	***Value_valid***: value /= Void **and then not** value.is_empty ***Current_system_id_valid***: current_system_id /= Void **and then not** current_system_id.is_empty ***Original_system_id_valid***: original_system_id /= Void **and then not** original_system_id.is_empty ***System_id_count_validity***: system_id_count > 0 ***Is_local_validity***: system_id_count > 1 **xor** (is_local and current_system_id.is_equal(original_system_id))

7 Resource Package

7.1 Overview

The `common.resource` package defines the structure and semantics of the general notion of an online resource which has been created by a human author, and consequently for which natural language is a factor. The package is illustrated in FIGURE 13.

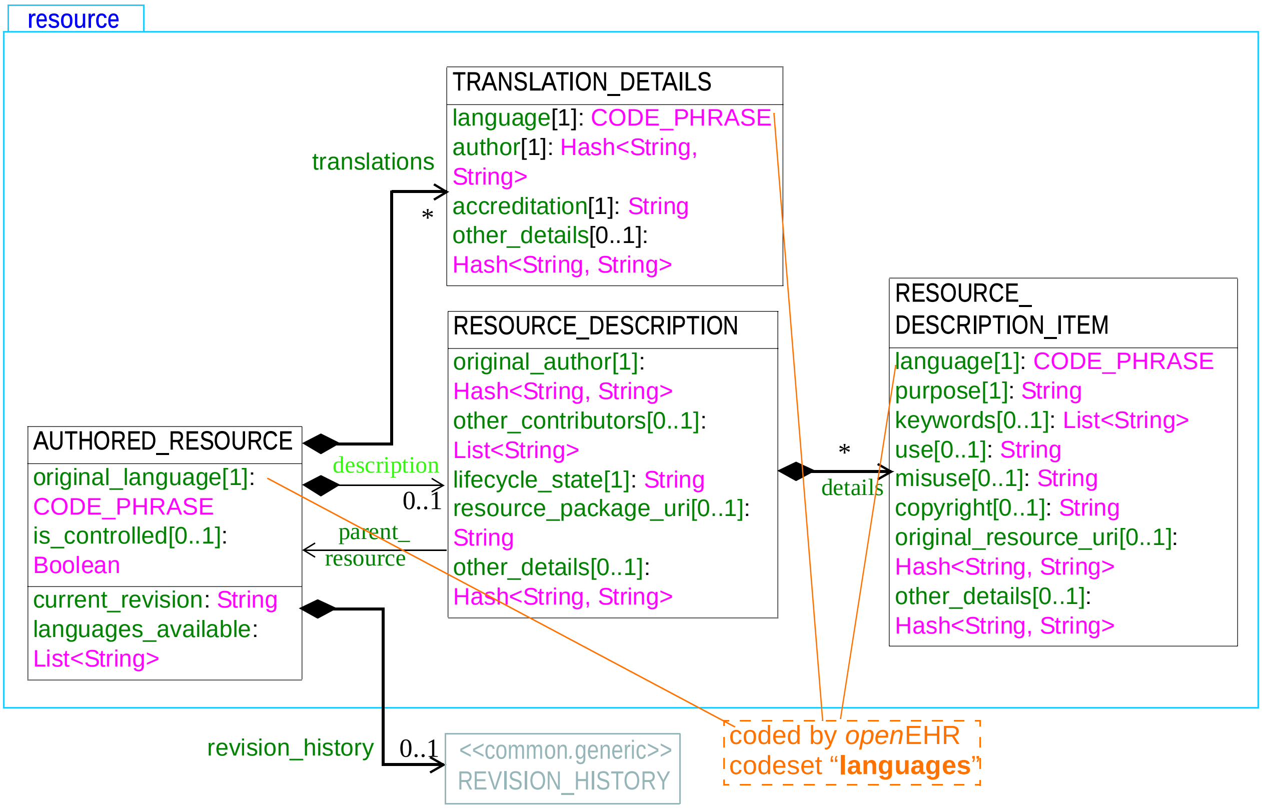

FIGURE 13 openehr.rm.common.resource Package

7.1.1 Natural Languages and Translation

Authored resources contain natural language elements, and are therefore created in some original language, recorded in the *orginal_language* attribute of the `AUTHORED_RESOURCE` class. Information about translations is included in the *translations* attribute, which allows for one or more sets of translation details to be recorded. A resource is translated by doing the following:

- translating every language-dependent element to the new language;
- adding a new `TRANSLATION_DETAILS` instance to *translations*, containing details about the translator, organisation, quality assurance and so on.
- any further translations to language-specific elements in a instances of descendent type of `AUTHORED_RESOURCE`.

The *languages_available* function provides a complete list of languages in the resource.

7.1.2 Meta-data

What is normally considered the 'meta-data' of a resource, i.e. its author, date of creation, purpose, and other descriptive items, is described by the `RESOURCE_DESCRIPTION` and

RESOURCE_DESCRIPTION_ITEM classes. The parts of this that are in natural language, and therefore may require translated versions, are represented in instances of the RESOURCE_DESCRIPTION_ITEM class. Thus, if a RESOURCE_DESCRIPTION has more than one RESOURCE_DESCRIPTION_ITEM, each of these should carry exactly the same information in a different natural language.

7.1.3 Revision History

When the resource is considered to be in a state where changes to it should be controlled, the *is_controlled* attribute is set to True, and all subsequent changes should have an audit trail recorded. Usually controlled resources would be managed in a versioned repository (e.g. implemented by CVS, Subversion or similar systems), and audit information will be stored somewhere in the repository (e.g. in version control files). The *revision_history* attribute defined in the AUTHROED_RESOURCE class is intended to act as a documentary copy of the revision history as known inside the repository, for the benefit of users of the resource. Given that resources in different places may well be managed in different kinds of repositories, having a copy of the revision history in a standardised form within the resource enables it to be used interoperably by authoring and other tools.

Every change to a resource committed to the relevant repository causes a new addition to the *revision_history.*

7.2 Class Definitions

7.2.1 AUTHORED_RESOURCE Class

CLASS	*AUTHORED_RESOURCE (abstract)*	
Purpose	Abstract idea of an online resource created by a human author.	
Attributes	Signature	Meaning
1..1	**original_language**: CODE_PHRASE	Language in which this resource was initially authored. Although there is no language primacy of resources overall, the language of original authoring is required to ensure natural language translations can preserve quality. Language is relevant in both the description and ontology sections.
0..1	**translations**: Hash <TRANSLATION_DETAILS, String>	List of details for each natural translation made of this resource, keyed by language. For each translation listed here, there must be corresponding sections in all language-dependent parts of the resource.
0..1	**description**: RESOURCE_DESCRIPTION	Description and lifecycle information of the resource.
0..1 (cond)	**revision_history**: REVISION_HISTORY	The revision history of the resource. Only required if *is_controlled* = True (avoids large revision histories for informal or private editing situations).

CLASS	*AUTHORED_RESOURCE (abstract)*	
1..1	**is_controlled**: `Boolean`	True if this resource is under any kind of change control (even file copying), in which case revision history is created.
Functions	**Signature**	**Meaning**
	current_revision: `String`	Current revision if *revision_history* exists else “(uncontrolled)”.
	languages_available: `Set<String>`	Total list of languages available in this resource, dervied from *original_language* and *translations*.
Invariant	***Original_language_valid***: original_language /= void **and then** language /= Void ***and then*** code_set(“languages”).has(original_language) ***Revision_history_validity***: is_controlled **implies** revision_history /= Void ***Current_revision_validity: not*** is_controlled ***implies*** current_revision.is_equal(“(uncontrolled)”)	

7.2.2 TRANSLATION_DETAILS Class

CLASS	TRANSLATION_DETAILS	
Purpose	Class providing details of a natural language translation.	
Attributes	**Signature**	**Meaning**
1..1	**language**: `CODE_PHRASE`	Language of translation
1..1	**author**: `Hash<String, String>`	Translator name and other demographic details
0..1	**accreditation**: `String`	Accreditation of translator, usually a national translator’s association id
0..1	**other_details**: `Hash<String, String>`	Any other meta-data
Invariant	***Language_valid***: language /= Void ***and then*** code_set(“languages”).has(language) ***Author_exists***: author /= Void	

7.2.3 RESOURCE_DESCRIPTION Class

CLASS	RESOURCE_DESCRIPTION	
Purpose	Defines the descriptive meta-data of a resource.	
Attributes	**Signature**	**Meaning**

CLASS	RESOURCE_DESCRIPTION	
1..1	**original_author**: `Hash<String, String>`	Original author of this resource, with all relevant details, including organisation.
0..1	**other_contributors**: `List<String>`	Other contributors to the resource, probably listed in “name <email>” form.
1..1	**lifecycle_state**: `String`	Lifecycle state of the resource, typically including states such as: `initial`, `submitted`, `experimental`, `awaiting_approval`, `approved`, `superseded`, `obsolete`.
1..1	**details**: `List<RESOURCE_DESCRIPTION_ITEM>`	Details of all parts of resource description that are natural language-dependent.
0..1	**resource_package_uri**: `String`	URI of package to which this resource belongs.
0..1	**other_details**: `Hash<String, String>`	Additional non language-senstive resource meta-data, as a list of name/value pairs.
1..1	**parent_resource**: `AUTHORED_RESOURCE`	Reference to owning resource.
Invariant	***original_author_validity***: original_author /= Void **and then not** original_author.is_empty ***details_exists***: details /= Void **and then not** details.is_empty ***language_validity***: details.for_all (d | parent_resource.languages_available.has(d.language)) ***Parent_resource_valid***: parent_resource /= Void and then parent_resource.description = Current	

7.2.4 RESOURCE_DESCRIPTION_ITEM Class

CLASS	**RESOURCE_DESCRIPTION_ITEM**	
Purpose	Language-specific detail of resource description. When a resource is translated for use in another language environment, each `RESOURCE_DESCRIPTION_ITEM` needs to be copied and translated into the new language.	
Attributes	**Signature**	**Meaning**
1..1	**language**: `CODE_PHRASE`	The localised language in which the items in this description item are written. Coded from *open*EHR Code Set “languages”.
1..1	**purpose**: `String`	Purpose of the resource.
0..1	**keywords**: `List<String>`	Keywords which characterise this resource, used e.g. for indexing and searching.
0..1	**use**: `String`	Description of the uses of the resource, i.e. contexts in which it could be used.
0..1	**misuse**: `String`	Description of any misuses of the resource, i.e. contexts in which it should not be used.
0..1	**copyright**: `String`	Optional copyright statement for the resource as a knowledge resource.
0..1	**original_resource_uri**: `Hash<String, String>`	URIs of original clinical document(s) or description of which resource is a formalisation, in the language of this description item; keyed by meaning.
0..1	**other_details**: `Hash<String, String>`	Additional language-senstive resource meta-data, as a list of name/value pairs.
Invariant	***Language_valid***: language /= Void ***and then*** code_set(“languages”).has(language) ***purpose_exists***: purpose /= Void **and then not** purpose.is_empty ***use_valid***: use /= Void **implies not** use.is_empty ***misuse_valid***: misuse /= Void **implies not** misuse.is_empty ***copyright_valid***: copyright /= Void **implies not** copyright.is_empty	

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

3 Hnìtynka P, Plášil F. *Distributed Versioning Model for MOF*. Proceedings of WISICT 2004, Cancun, Mexico, A volume in the ACM international conference proceedings series, published by Computer Science Press, Trinity College Dublin Ireland, 2004.

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

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