openEHR

Release 1.0.1

The *open*EHR Reference Model

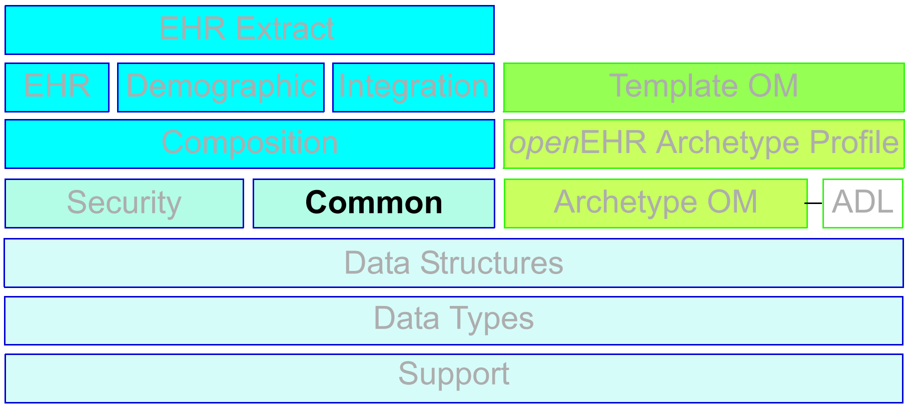
Common Information Model

Editors: {T Beale, S Heard}[a], {D Kalra, D Lloyd}[b]		
Revision: 2.1.0	*Pages:* 91	*Date of issue:* 08 Apr 2007

a. Ocean Informatics

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

Keywords: common, EHR, reference model, openehr



Copyright Notice

Amendment Record

Issue	Details	Raiser	Completed
	RELEASE 1.0.1		
2.1.0	**CR-000209**: Minor changes to correctly define AUTHORED_RESOURCE.*current_revision*. Functions added to REVISION_HISTORY; AUTHORED_RESOURCE.*current_revision* postcondition added.	Y S Lim	08 Apr 2007
	CR-000206: Change LOCATABLE.*item_at_path* to return ANY	H Frankel	
	CR-000200: Correct Release 1.0 typographical errors. Add missed invariant in VERSION to restrict *contribution*.type to "CONTRIBUTION". Fix pre- and post-conditions in LOCATABLE functions.	T Beale	
	Fix errors in timezone max/min values and invariants.	T Cook	
	CR-000203: Release 1.0 explanatory text improvements. Move Explanatory material on configuration management and versioning to Architecture Overview.	T Beale E Sundvall	
	CR-000202: Correct minor errors in VERSION.*preceding_version_id*. Rename *preceding_version_id* to *preceding_version_uid*. Add *preceding_version_uid* invariant to VERSION<T>.	T Beale	
	CR-000197: Change LOCATABLE.*uid* to HIER_OBJECT_ID	H Frankel	
	CR-000214: Changes to VERSION preparatory to EHR Extract upgrade. Added *lifecycle_state* to VERSION<T>, extra functions on VERSIONED_OBJECT<T>. Corrected and added *commit* functions to VERSIONED_OBJECT. Added ATTESTATION.*attested_view* (conforms to CEN EN13606-1).	H Frankel S Heard T Beale	
	CR-000212: Allow VERSION.*data* to be optional to enable logical deletion.	T Beale	
	CR-000130: Correct security details in LOCATABLE and ARCHETYPED classes. Remove ARCHETYPED.*access_control*.	T Beale	
	CR-000219: Use constants instead of literals to refer to terminology in RM.	R Chen	
	CR-000231: Change RESOURCE_DESCRIPTION.*details* from List to Hash.	R Chen	
	CR-000235: Make attestation-only commit require a Contribution.	A Patterson	
	CR-000239: Add common parent type of OBJECT_VERSION_ID and HIER_OBJECT_ID.	H Frankel	
	CR-000243: Add *template_id* to ARCHETYPED class.	T Beale	
	CR-000244: Separate LOCATABLE path functions into PATHABLE class.	T Beale H Frankel	
	CR-000166: Add viewable form of document to COMPOSITION	S Heard	
	CR-000246: Correct *open*EHR terminology rubrics.	B Verhees M Forss	
	RELEASE 1.0		

Issue	Details	Raiser	Completed
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. Rename REVISION_HISTORY_ITEM.*revision* to *version_id*, and change type to OBJECT_VERSION_ID.	R Chen 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 patterns 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 Architecture Overview
- 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-1.0.1/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.

2 Overview

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

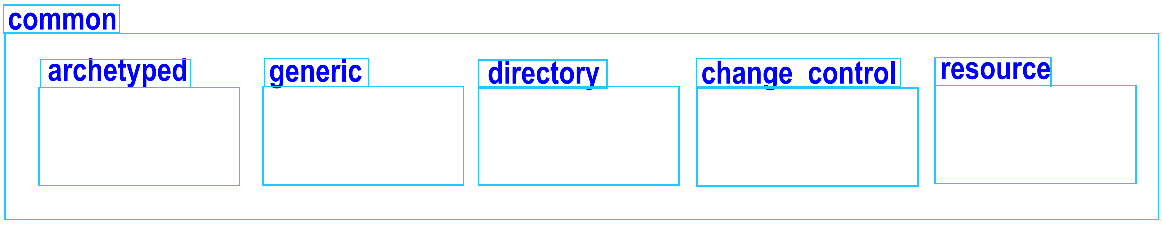


FIGURE 1 rm.common Package

The `archetyped` package 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 forming 'analysis patterns' 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 logical 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 defines the core types `PATHABLE`, `LOCATABLE`, `ARCHETYPED`, and `LINK`. It is illustrated in FIGURE 2.

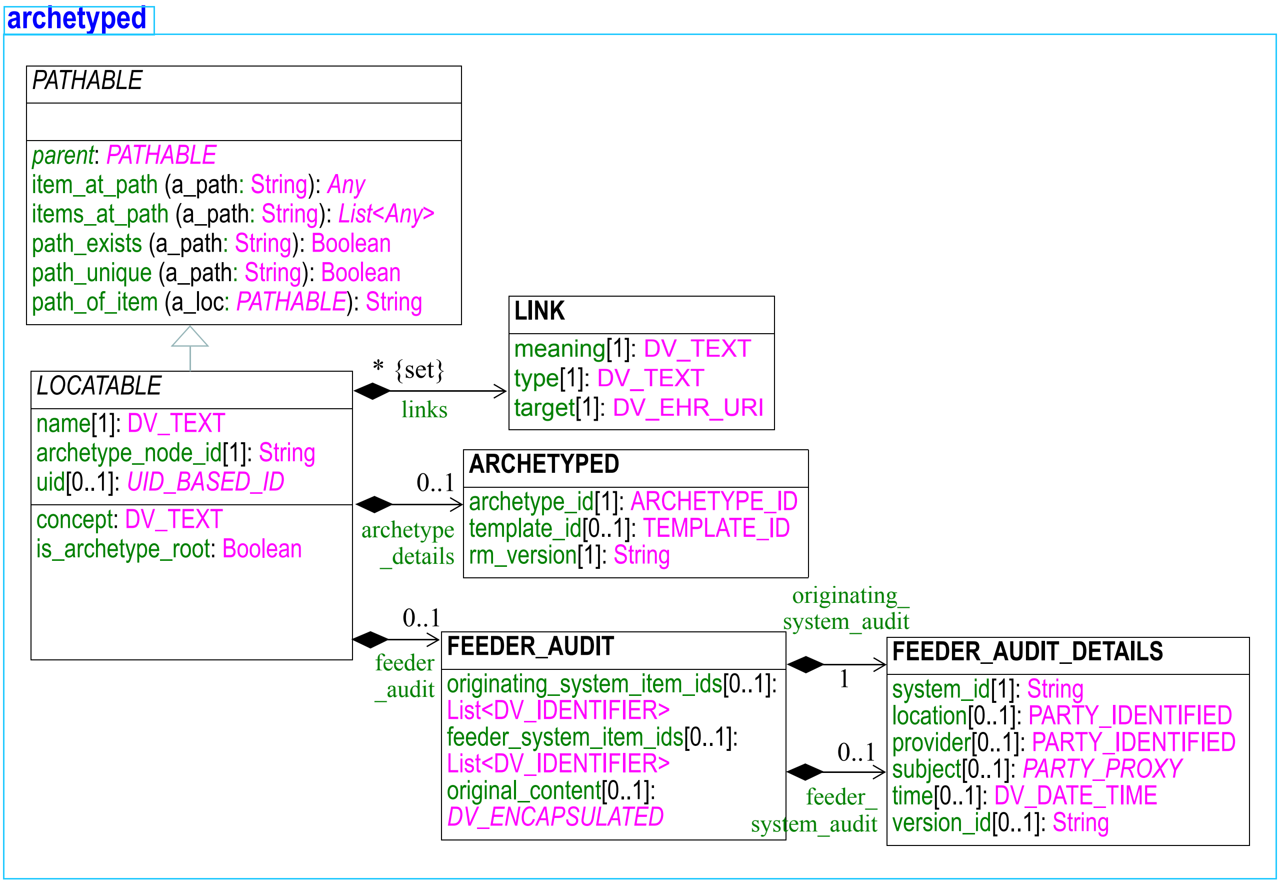


FIGURE 2 rm.common.archetyped Package

3.1.1 The PATHABLE Class

The `PATHABLE` class defines the pathing capabilities used by nearly all classes in the *open*EHR reference model, mostly via inheritance of `LOCATABLE`. The defining characteristics of `PATHABLE` objects are that they can locate child objects using paths, and they know their *parent* object in a compositional hierarchy. The *parent* feature is defined as abstract in the model, and may be implemented in any way convenient.

A number of functions provide the path functionality, of which `item_at_path()` and `items_at_path()` are the key functions. The former returns an item corresponding to a unique path, i.e. a path that resolves against the data structure to a single node. The latter returns a list of items corresponding to a non-unique path. These functions can be used safely using the following pattern, but can also be used without checking the validity of paths, if this is known *a priori* in the code anyway.

```
    if path_exists(a_path) {
            if path_unique(a_path) {
                  x := item_at_path(a_path)
```

```
            // process one item
        }
        else {
            list_of_x := items_at_path(a_path)
            //iterate the list
        }
    }
```

3.1.2 The LOCATABLE Class

Most classes in the *open*EHR reference model inherit from the LOCATABLE class, which defines the idea of 'locatability in an archetyped structure'. LOCATABLE defines 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 only exception is at archetype root points in data, where *archetype_node_id* carries the archetype identifier in string form rather than an interior node id from an archetype. LOCATABLE also provides the attribute *archetype_details*, which is non-Void for archetype root points in data, and carries meta-data relevant to root points. 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 in a LOCATABLE instance 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 might 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 on the node in question, unless explicitly set otherwise.

Unique Node Identification

LOCATABLE descendants may have a *uid*. In the current *open*EHR architecture, uids are not needed to identify data nodes, since paths are used to reference all nodes inside top-level structures (i.e. COMPOSITIONs etc). Accordingly all references between parts of an EHR are represented in terms of LOCATABLE_REFs or DV_EHR_URIs (the former is a reference to an OBJECT_VERSION_ID with a path appended; the latter is the stringified URI form). This would allow for example, one Entry to reference the serum sodium value in another Entry in version 2 of a Versioned Composition for a laboratory test on 12/Apr/2004. The *uid* attribute will usually be empty in most EHR data in most *open*EHR EHR systems.

The exception is the top-level types such as COMPOSITION, EHR_STATUS, PARTY etc for which it is recommended to set the *uid* value to a copy of the *uid* attribute of the owning VERSION object. This enables easy identification of standalone top-level objects in a serialised form

Another use for LOCATABLE.*uid* is in EHR Extracts, which contain serialised expressions of EHR content. In an Extract, the uid could be set on some or all nodes to a value generated by concatenating the uid of the enclosing Version object (i.e. VERSION.*uid*) and the unique runtime path to the particular node. This may be useful to the receiver system for the purpose of referencing particular data nodes when communicating to the sender, or another system. This use of uids is not however mandatory, since for each node in an Extract item, the uid can be generated at any time (including at the receiver system).

Note: some classes in the *open*EHR architecture that do not inherit from LOCATABLE but require a uid, such as VERSIONED_OBJECT, VERSION etc, explicitly define their own *uid* attribute.

3.1.3 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 that have been transformed into *open*EHR form and committed to the system. There are a number of 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 immediate 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;
- allow the inclusion of the source content either as a link or inline.

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 an *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;

original content store: some EHR systems may have an associated persistent repository of content as received from external systems, e.g. a message or document database.

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.

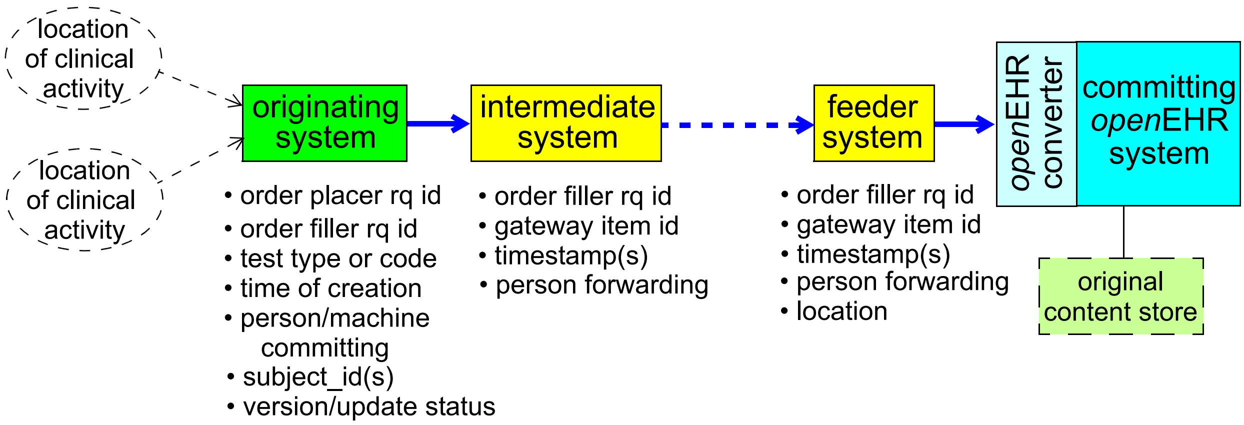


FIGURE 3 Abstract model of feeder chain

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 task 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 of 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. Alternative conversion approaches 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 Clusters and Elements (see *open*EHR Data Structure IM or CEN EN13606), but no Entries, 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".

Original Content

The features of the model described so far allow accurate referencing of content as it is known in source systems and intermediate feeder systems. A further feature of the FEEDER_AUDIT class, the *original_content* attribute allows the original content item itself to be either included inline or pointed to. If a link is used, the usual situation is that the content is in a store associated with the receiving system, such as a message or document database. The content could also be included inline. Since the *original_content* link is on a FEEDER_AUDIT object, more than one can be used within the same generated Composition if required. It may be deemed preferable to attach only a single link at the top node, i.e. the Composition node, since this establishes basic equivalent between the whole Composition and the whole document or message.

3.2 Class Descriptions

3.2.1 Class PATHABLE

CLASS	***PATHABLE (abstract)***	
Purpose	Abstract parent of all classes whose instances are reachable by paths, and which know how to locate child object by paths. The *parent* feature may be implemented as a function or attribute.	
Abstract	**Signature**	**Meaning**
0..1	**parent**: *PATHABLE*	Parent of this node in compositional hierarchy.
Functions	**Signature**	**Meaning**
	path_of_item (an_item: *PATHABLE*): String ***require*** **item_valid**: an_item /= Void	The path to an item relative to the root of this archetyped structure.
	item_at_path (a_path: String): *Any* ***require*** a_path /=Void **and then** path_unique(a_path) ***ensure*** Result /= Void	The item at a path (relative to this item); only valid for unique paths, i.e. paths that resolve to a single item.
	items_at_path (a_path: String): List<*Any*> ***require*** path /=Void **and then not** path_unique(a_path) ***ensure*** Result /= Void	List of items corresponding to a non-unique path.
	path_exists (a_path: String): Boolean ***require*** **path_valid:** a_path /= Void **and then** not a_path.is_empty	True if the path exists in the data with respect to the current item.
	path_unique (a_path: String): Boolean ***require*** **path_valid:** a_path /= Void **and then** path_exists(a_path)	True if the path corresponds to a single item in the data.
Invariant		

3.2.2 Class LOCATABLE

CLASS	*LOCATABLE (abstract)*	
Purpose	Root class of all information model classes that can be archetyped.	
CEN	RECORD_COMPONENT	
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.	
Inherit	PATHABLE	
Attributes	**Signature**	**Meaning**
0..1	**uid**: *UID_BASED_ID*	Optional globally unique object identifier for root points of archetyped structures.
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.

CLASS	LOCATABLE (abstract)	
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.
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**
1..1	**is_archetype_root**: `Boolean`	True if this node is the root of an archetyped structure.
0..1	**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	***Name_valid***: name /= Void ***Links_valid***: links /= Void **implies not** links.empty ***Archetyped_valid***: is_archetype_root ***xor*** archetype_details = Void ***Archetype_node_id_valid***: archetype_node_id /= Void **and then not** archetype_node_id.is_empty	

3.2.3 ARCHETYPED Class

CLASS	ARCHETYPED
Purpose	Archetypes act as the configuration basis for the particular structures of instances defined by the reference model. To enable archetypes to be used to create valid data, key classes in the reference model act as "root" points for archetyping; accordingly, these classes have the archetype_details attribute set. An instance of the class `ARCHETYPED` contains the relevant archetype identification information, allowing generating archetypes to be matched up with data instances
GEHR	`G1_ARCHETYPED`
Synapses/ SynEx	The SynEx approach does not distinguish between multiple layers of archetypes; hence an 'archetype' covers all information in an entire composition. Consequently, there is only one place where archetype identifiers in the *open*EHR sense are used (at the top); all other archetype identifiers are equivalent to the *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.

CLASS	ARCHETYPED	
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 2005 revision of EN 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	**template_id**: `TEMPLATE_ID`	Globally unique template identifier, if a template was active at this point in the structure. Normally, a template would only be used at the top of a top-level structure, but the possibility exists for templates at lower levels.
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_valid***: archetype_id /= Void ***rm_version_valid***: rm_version /= Void ***and then not*** rm_version.is_empty	

3.2.4 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.
MisUse	Links should be between archetyped structures 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 `ENTRY`s, `SECTION`s, `COMPOSITION`s and so on.

CLASS	LINK	
CEN	The Link Item class is a simplified form of the Synapses Link Item, permitting links to be established but with limited labelling and no representation for importance.	
Synapses	The Link Item class provides the means to link any arbitrary parts of a single EHR, for the overall linkage network to be labelled and revised, and for each direct link to be labelled explicitly. An importance attribute provides guidance on how links should be handled if only part of a linkage network is requested by a client process.	
GEHR	n/a	
HL7v3	The ACT_RELATIONSHIP class in some cases appears to correspond to LINK.	
Attributes	**Signature**	**Meaning**
1..1	**meaning**: DV_TEXT	Used to describe the relationship, usually in clinical terms, such as "in response to" (the relationship between test results and an order), "follow-up to" and so on. Such relationships can represent any clinically meaningful connection between pieces of information. 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".
1..1	**type**: DV_TEXT	The *type* attribute is used to indicate a clinical or domain-level meaning for the kind of link, for example "problem" or "issue". If type values are designed appropriately, they can be used by the requestor of EHR extracts to categorise links which must be followed and which can be broken when the extract is created.
1..1	**target**: DV_EHR_URI	The logical "to" object in the link relation, as per the linguistic sense of the *meaning* attribute.
Invariant	***Meaning_valid***: meaning /= Void ***Type_valid***: type /= Void ***Target_valid***: target /= Void	

3.2.5 FEEDER_AUDIT Class

CLASS	FEEDER_AUDIT
Purpose	Audit and other meta-data for systems in the feeder chain.

CLASS	FEEDER_AUDIT	
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.
0..1	**original_content**: *DV_ENCAPSULATED*	Optional inline inclusion of or reference to original content corresponding to the *open*EHR content at this node. Typically a URI reference to a document or message in a persistent store associated with the EHR.
Invariants	***Originating_system_audit_valid***: originating_system_audit /= Void	

3.2.6 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.

CLASS	FEEDER_AUDIT_DETAILS	
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 patterns in the domain of health (and most likely other domains), such as ‘participation’ and ‘attestation’. The generic cluster is illustrated in FIGURE 4.

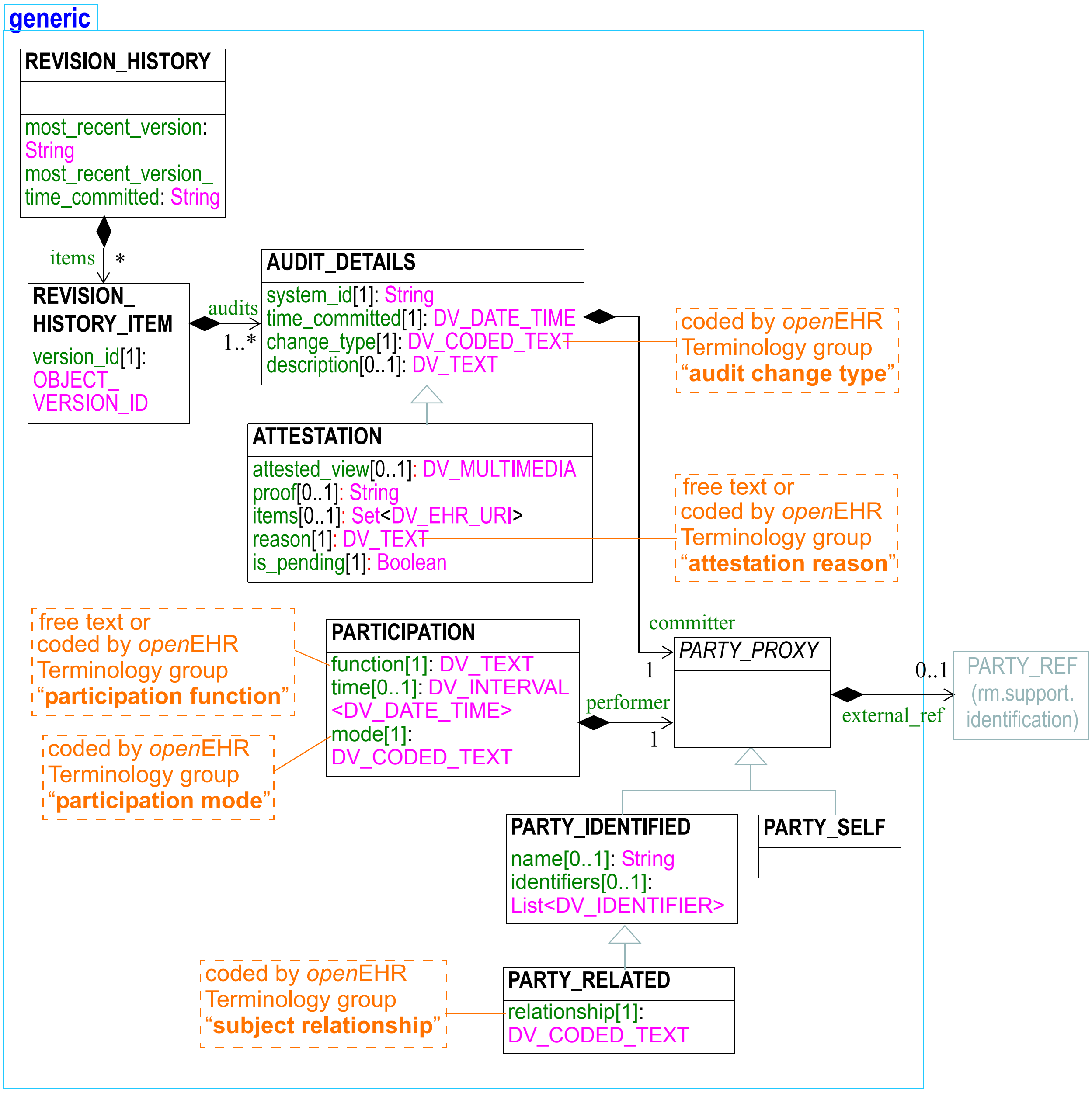


FIGURE 4 rm.common.generic Package

4.2 Design Principles

4.2.1 Referring to Demographic Entities

There are two ways to refer to a demographic 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, consisting of 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.

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, it should be referenceable;
- 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. Each uses a PARTY_SELF object but with varying usage of the *external_ref* attribute, and are as follows.

- The *external_ref* attribute is not set on any instances of PARTY_SELF, i.e. nowhere in the EHR. 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.*ehr_id* and the patient demographic/PMI identifier. This approach is more likely in more open data sharing environments.
- The *external_ref* attribute is set once only in EHR_STATUS.*subject*. Since the EHR_STATUS 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.

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, the 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 *open*EHR information 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 *version_id* is included to indicate which 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. The contents of an ATTESTATION are as follows:

- the identity of the attesting party (AUDIT_DETAILS.*committer*);
- the date and time of the action of attestation (AUDIT_DETAILS.*time_committed*);
- references to items in the record being attested to (ATTESTATION.*items*); if this list is empty, the attestation is for the entire object (usually the content of an ORIGINAL_VERSION) to which the attestation is attached, otherwise the list must contain a set of paths to items within the item to which the attestation is attached;
- an optionally coded reason for attestation (ATTESTATION.*reason*);
- an optional literal view of the the content attested, e.g. a binary screen image;
- a proof of attestation in the form of a digital signature by the attesting party.

The digital signature, if present, is generated using the IETF RFC 2440 (openPGP)[1] standard as, according to the process shown in FIGURE 5.

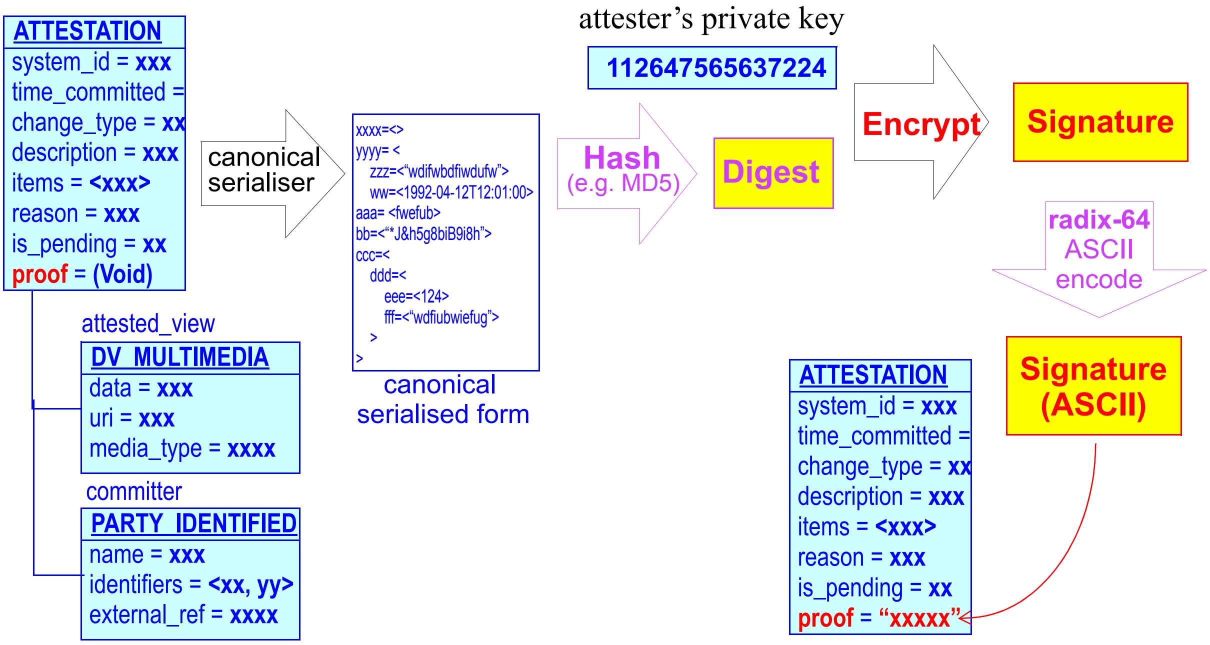


FIGURE 5 Attestation signature generation (using openPGP)

In this process, the attestation object is serialised into a canonical text form, and then hashed to create a digest. A digital signature is created from the hash, using the user's private key. The result is then radix-64 encoded to create an ASCII string so as to remove or reduce potential problems with subsequent communication. The openPGP standard ensures that the transformations and algorithms used to create the signature are indicated within it (i.e. the signature is self-describing).

The serialisation process works by the simple rule of serialising the entire Attestation object (note that the *proof* attribute will be Void at this point) into an agreed XML, dADL or other text format, then applying the subsequent transformations to the serialised data, then writing the digest result back into the *proof* attribute.

To Be Determined: The exact serialisation is not yet defined by *open*EHR, but dADL might be preferred since it has an unambiguous encoding of object struc-

1. See http://www.ietf.org/rfc/rfc2440.txt

tures, whereas XML libraries generate different XML from the same objects.

Normally the list of items being attested 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 a *commit_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 signing occurs, 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. Thus, the common situation in which content is committed to the record and needs later review and signing by a senior person will cause the creation of two ATTESTATION objects.

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_valid*** name /= Void **or** identifiers /= Void **or** external_ref /= Void ***Name_valid***: name /= Void **implies not** name.is_empty ***Identifiers_valid***: identifiers /= Void **implies not** identifiers.is_empty	

4.3.4 PARTY_RELATED Class

CLASS	PARTY_RELATED	
Purpose	Proxy type for identifying 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**

CLASS	PARTY_RELATED	
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".
Invariants	***Relationship_valid***: relationship /= Void **and then** terminology(Terminology_id_openehr). has_code_for_group_id(Group_id_subject_relationship, 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.

CLASS	PARTICIPATION
Invariant	***Performer_valid***: performer /= Void ***Function_valid***: function /= Void **and then** function.generating_type.is_equal("DV_CODED_TEXT") **implies** terminology(Terminology_id_openehr).has_code_for_group_id(Group_id_participation_function, function.defining_code) ***Mode_valid***: mode /= Void **and** terminology(Terminology_id_openehr).has_code_for_group_id(Group_id_participation_mode, mode.defining_code)

4.3.6 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. Ideally this is a machine- and human-processable identifier, but it may not be.
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_valid***: system_id /= Void **and then not** system_id.is_empty ***Committer_valid***: committer /= Void ***Time_committed_valid***: time_committed /= Void ***Change_type_valid***: change_type /= Void **and then** terminology(Terminology_id_openehr).has_code_for_group_id(Group_id_audit_change_type, change_type.defining_code)	

4.3.7 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	**attested_view**: `DV_MULTIMEDIA`	Optional visual representation of content attested e.g. screen image.
0..1	**proof**: `String`	Proof of attestation.
0..1	**items**: `Set <DV_EHR_URI>`	Items attested, expressed as fully qualified runtime paths to the items in question. 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_valid***: items /= Void **implies not** items.is_empty ***Reason_valid***: reason /= Void **and then** (reason.generating_type.is_equal("DV_CODED_TEXT") **implies** terminology(Terminology_id_openehr).has_code_for_group_id(Group_id_attestation_reason, reason.defining_code))	

4.3.8 REVISION_HISTORY Class

CLASS	REVISION_HISTORY	
Purpose	Defines the notion of a revision history of audit items, each associated with the version for which that audit was committed. The list is in most-recent-first order.	
Attributes	**Signature**	**Meaning**
1..1	**items**: `List <REVISION_HISTORY_ITEM>`	The items in this history in most-recent-last order.
Function	**Signature**	**Meaning**

CLASS	REVISION_HISTORY	
	most_recent_version: `String` ***ensure*** Result.is_equal (items.last.version_id.value)	The version id of the most recent item, as a String.
	most_recent_version_time_committed: `String` ***ensure*** Result.is_equal (items.last.audits.first.time_committed.value)	The commit date/time of the most recent item, as a String.
Invariants	***Items_valid***: 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..1	**audits**: `List<AUDIT_DETAILS>`	The audits for this revision; there will always be at least one commit audit (which may itself be an `ATTESTATION`), there may also be further attestations.
1..1	**version_id**: `OBJECT_VERSION_ID`	Version identifier for this revision.
Invariants	***Audit_valid***: audits /= Void **and then not** audits.is_empty ***Version_id_valid***: version_id /= Void	

5 Directory Package

5.1 Overview

The directory package is illustrated in FIGURE 6. It provides a simple abstraction of a versioned folder structure. The VERSIONED_FOLDER class is the binding of VERSIONED_OBJECT<T> to the class FOLDER, i.e. it is a VERSIONED_OBJECT<FOLDER>. This means that each of its versions is a Folder structure rather than a single Folder. 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 contains more FOLDERs 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 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_COMPOSITIONs for example, the folders might be used to represent episodes and at the same time problem groups.

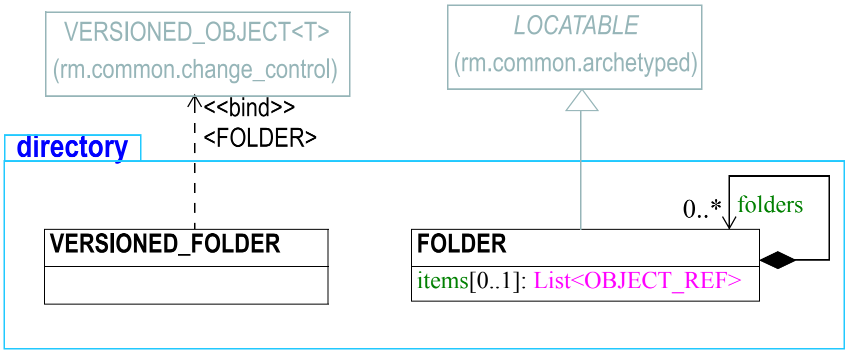


FIGURE 6 common.directory Package

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 (e.g. within the EHR):

```
/folders[hospital episodes]/items[1]
/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.is_empty	

6 Change Control Package

6.1 Overview

As described in the Architecture Overview document, formal version control and change management are used in *open*EHR to support the construction of EHR and other repositories requiring the properties of consistency, indelibility, traceability and distributed sharing. The `change_control` package supplies the formal specification of these features in *open*EHR.

FIGURE 7 illustrates the *open*EHR model of a Versioned object, and its constituent Versions. In this model, an instance of the class `VERSIONED_OBJECT<T>` provides the versioning facilities for one versioned item and is often referred to as a 'version container'. Although any kind of data can be versioned according to the model presented here, the use of versioning in *open*EHR is limited to 'top-level structures', such as EHR Compositions and Party objects in a demographic system.

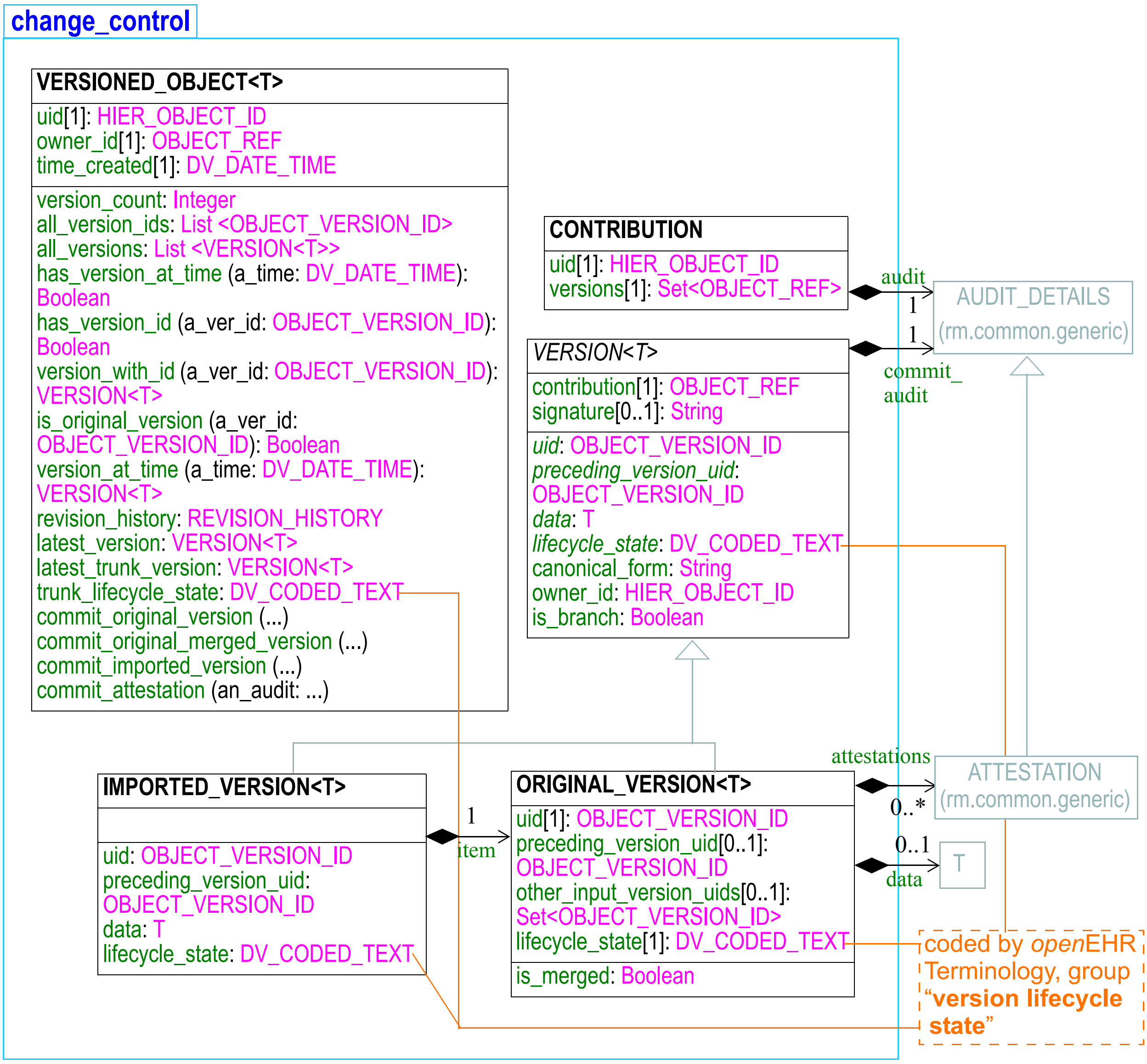


FIGURE 7 rm.common.change_control Package

FIGURE 8 illustrates a single VERSIONED_OBJECT containing a number of VERSIONs. Although the figure implies physical containment of Versions by a Versioned object, this is only one possible implementation. Other implementations (e.g. using orthodox relational structures) might use references, separate compressed copies, or any other mechanism.

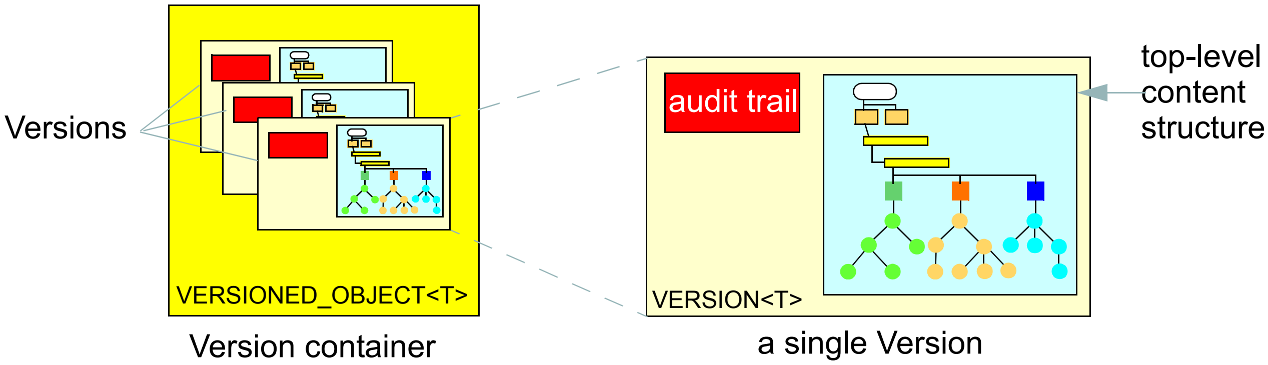


FIGURE 8 Version-control structures

6.2 Basic Semantics

6.2.1 Typing

The classes VERSIONED_OBJECT<T>, VERSION<T>, ORIGINAL_VERSION<T> and IMPORTED_VERSION<T> are generic classes, with the generic parameter type T being the type of the data. This ensures that all versions in a given VERSIONED_OBJECT are of the same type, such as COMPOSITION, FOLDER, or PARTY and that the version container itself is properly typed.

6.2.2 Versioned Objects

Each VERSIONED_OBJECT has a unique identifier recorded in the *uid* attribute (a HIER_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 data in a VERSIONED_OBJECT are in the form of a collection of instances of the two VERSION<T> subtypes, and are available only via the functional interface of VERSIONED_OBJECT. How the representation of this collection is implemented inside the VERSIONED_OBJECT is not defined by this specification, only the form of any given Version is. Implementations of VERSIONED_OBJECT might range from the simple (all versions stored as full copies in a list) to a sophisticated compressed versioning approach as used in software file version control and some object databases. (The persistent data format of implementations of VERSIONED_OBJECT developed by different organisations will in general be incompatible. For purposes of sharing, an interoperable expression of VERSIONED_OBJECT is defined by the X_VERSIONED_OBJECT class in the EHR Extract IM.)

6.2.3 Version and its Subtypes

Within a Versioned object, each version is an instance of a subtype of the class VERSION<T>. The abstract VERSION class defines the generic notion of a version containing some *data*, that has been committed to the repository as a member of a Contribution. Accordingly, it records the Contribution in the *contribution* attribute and the audit in *commit_audit*. A Version also knows its position in the version tree within the container. It has a version identifier, *uid*, and knows on which version in the tree it was based (i.e. what version was “checked out” to create the current version), *preceding_version_id* (Void if it is the first version). Both of these identifiers are globally unique (see `support.identification` package). These properties are abstract in the `VERSION` class, since they are defined as being stored or computed respectively in its subtypes.

All Versions in a given version container have a uid that includes the uid of the container; in other words, the uid of a Version is its container's uid plus further version identification for that particular version with respect to others in the same container. The `VERSION`.*owner_id* function extracts the uid of the owning `VERSIONED_OBJECT` from the *uid* of the `VERSION`.

The `VERSION` class has two subtypes. The first, `ORIGINAL_VERSION<T>`, represents a Version created with original content (stored form of *data* property) at the time of creation (including from non-*open*EHR local feeder systems), and potentially attested (signed). It includes as attributes the current version (*uid*) and the preceding version (*preceding_version_uid*). It also knows the lifecycle state of its content. If it was the result of a merge (see Version Merging on page 52) of versions other than the preceding version, the identifiers of these versions will be recorded in the attribute *other_input_version_uids*. All instances of `VERSION<T>` in non-distributed *open*EHR systems will be instances of `ORIGINAL_VERSION<T>`. The `ORIGINAL_VERSION` is also the unit of copying in a distributed environment.

The second subtype is `IMPORTED_VERSION<T>`, and acts as a wrapper of an `ORIGINAL_VERSION<T>`. It has its own *contribution* and *commit_audit* (inherited from `VERSION<T>`), and contains the original version being imported in its *item* attribute. Its *uid* and *preceding_version* are defined as functions, returning the corresponding attribute values from the wrapped `ORIGINAL_VERSION` object (in other words, an `IMPORTED_VERSION` does not have its own version identifier distinct from the version it is wrapping). The semantics of importing are described below in Copying on page 50. FIGURE 9 illustrates typical arrangements of `ORIGINAL_VERSION` and `IMPORTED_VERSION` objects within `VERSIONED_OBJECTs`, in turn within an EHR (if this is an EHR system), ultimately within an identified system. The two `VERSIONED_OBJECTs` are shown representing "medications" and "problem list", to give some idea of correspondence of versioning structures to logical data. Star icons represent digital signatures.

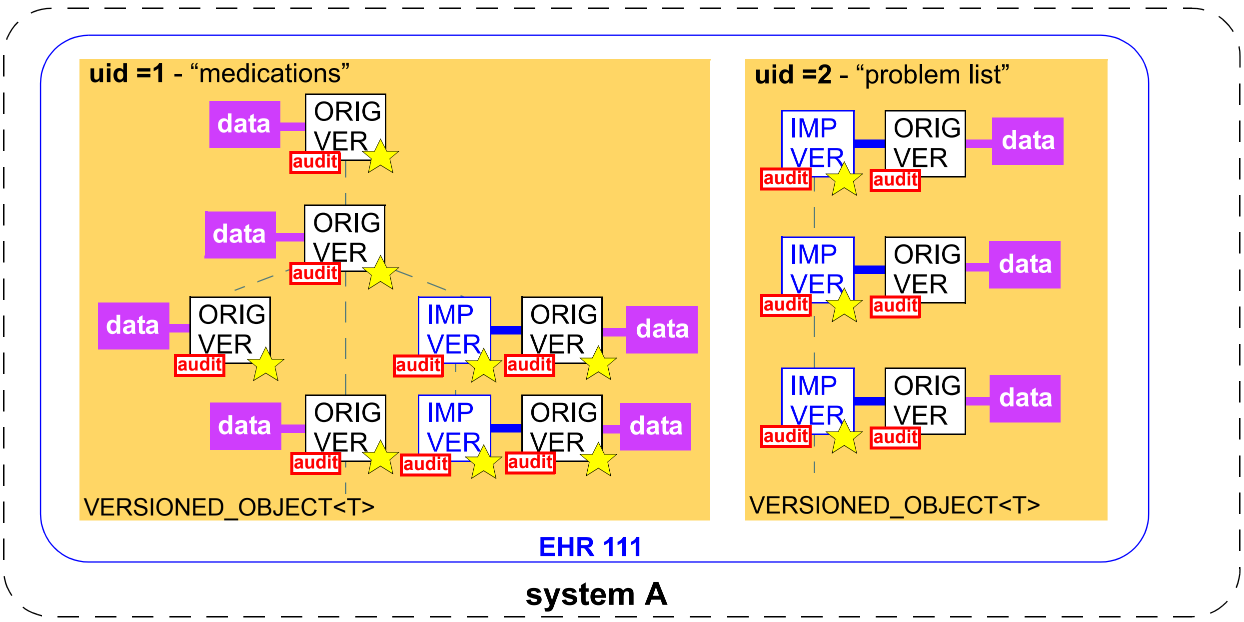


FIGURE 9 Instance view of versioned data

6.2.4 The "Virtual Version Tree"

An underlying design concept of the versioning model defined here is known as the 'virtual version tree'. The idea is simple in the abstract. Information is committed to a repository (such as an EHR) in lumps, each lump being the 'data' of one Version. Each Version has its place within a version tree, which in turn is maintained inside a Versioned object. The virtual version tree concept means that any given Versioned object may have numerous copies in various systems, and that the creation of versions in each is done in such a way that all versions so created are in fact compatible with the 'virtual' version tree resulting from the superimposition of the version trees of all copies. This is achieved using simple rules for version identification, described below, and is done to facilitate data sharing. Two very common scenarios are served by the virtual version tree concept:

- longitudinal data that stands as a proxy for the state or situation of the patient such as "Medications" or "Problem list" (persistent Compositions in *open*EHR) is created and maintained in one or more care delivery organisations, and shared across a larger number of organisations;
- some EHRs in an EHR server in one location are mirrored into one or more other EHR servers (e.g. at care providers where the relevant patients are also treated); the mirroring process requires asynchronous synchronisation between servers to work seamlessly, regardless of the location, time, or author of any data created.

The *uid* attribute of the class `VERSIONED_OBJECT<T>` is in fact the uid of the virtual version tree for a given logical item (such as the "problem list" of a certain patient) - that is to say, the uid will be the same in all copies of the same Versioned object in a distributed system.

The versioning scheme used in *open*EHR guarantees that no matter where data are created or copied, there are no inconsistencies due to sharing, and that logical copies are explicitly represented. This is achieved by the design of Version identifiers.

6.2.5 Contributions

Since a versioned repository (i.e. a collection of `VERSIONED_OBJECT`s) is by definition indelible, all *logical* changes including deletions, additions, modifications (including error corrections and content changes), importing and attestations of existing items, are achieved by physically committing new Versions, or for attestations, new Attestation objects to existing Versions. Each logical type of change is achieved as follows:

- *addition of new item*: a new `VERSIONED_OBJECT` is created with a first `ORIGINAL_VERSION` whose *data* is the new item; the `ORIGINAL_VERSION`.*commit_audit*.*change_type* is set to the code for 'creation'
- *deletion of existing item*: a new `ORIGINAL_VERSION` whose *data* attribute is set to Void is added to an existing `VERSIONED_OBJECT`; the `ORIGINAL_VERSION`.*commit_audit*.*change_type* is set to the code for 'deleted';
- *modification of existing item*: a new `ORIGINAL_VERSION` whose *data* contains the updated form of the item content is added to an existing `VERSIONED_OBJECT`;
    - if the change is logically a correction (e.g. of wrongly entered data), the `ORIGINAL_VERSION`.*commit_audit*.*change_type* is set to the code for 'amendment';
    - if the change is logically a change, addition etc to the content, the `ORIGINAL_VERSION`.*commit_audit*.*change_type* is set to the code for 'modification';

- *import of item*: a new IMPORTED_VERSION is created, incorporating the received ORIGINAL_VERSION; the IMPORTED_VERSION.*commit_audit.change_type* is set to the code for 'creation'.
- *attestation of item*: a new ATTESTATION is added to the *attestations* list of an existing ORIGINAL_VERSION; the ATTESTATION.*commit_audit.change_type* is set to the code for 'attestation'.

In a typical application situation, one or more of the above changes may be committed to a repository as a Contribution. For example during a patient encounter, the following might occur:

- *addition*: a new Composition is created recording the Observations (e.g. physical examination), etc that are made during the Encounter;
- *modification*: the Composition containing the current medications list is updated, due to a prescription being given during the encounter.

These two changes together constitute a logical 'change-set', and would typically be included in the one Contribution. In general, there might be any combination of the logical change types in a single commit by an application, corresponding to a single real-world business event, such as a GP Encounter, although attestations, deletions and corrections will usually be the only change within a Contribution. In every case, regardless of the combination, a CONTRIBUTION object will be created, listing the affected VERSION objects, and including its own audit object.

The list of all Contribution objects for a version repository (such as an EHR) provides a complete history of the change-sets made to the repository and is the basis for performing 'rollback' to access previous informational states of the EHR. Conversely, each Version object contains a reference to the Contribution that caused it to be created.

6.2.6 Committal and Audits

Audits are recorded in the form of instances of the class AUDIT_DETAILS (common.generic package), which defines a set of attributes which form an audit trail, namely *system_id*, *committer*, *time_committed*, *change_type*, and *description* or its subtype ATTESTATION, which adds a number of other attributes (see below). When an ORIGINAL_VERSION instance is created locally, the *commit_audit* attribute contains an audit object recording the local act of committal.

However, if the Version being committed does not correspond to local data creation, but instead contains a copy of an ORIGINAL_VERSION originally created and commited elsewhere, it is committed locally as an instance of the IMPORTED_VERSION class. Both the *contribution* and *commit_audit* of the latter object correspond to the local act of committal, while the knowledge of the original Contribution and committal are retained inside the wrapped ORIGINAL_VERSION instance. Original versions can be copied any number of times; in each system into which they are imported, an IMPORTED_VERSION is created as a wrapper.

This simple scheme ensures that the audit from initial creation - which is the clinically meaningful audit - is preserved no matter how many times the Version is copied to other systems; it also ensures that from the point of view of the version container, the local commit audit and Contribution always correspond to the local act of committal.

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) should be 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.

The *time_committed* attribute in both the Contribution and Version audits should reflect the time of committal to an EHR server, i.e. the time of availability to other users in the same system. It should therefore be computed on the server in implementations where the data are created in a separate client context.

In terms of database management, Contributions are similar to nested transactions. An attempt to commit a Contribution should only succeed if each Version and/or Attestation in the Contribution is committed successfully.

6.2.7 Digital Signature

At the time of committal of a Version, a digital signature of the object can be made. In this process, a Version object (an `ORIGINAL_VERSION` or `IMPORTED_VERSION`) is serialised into canonical form which is then hashed to produce a digest. If public key or equivalent infrastructure is in place so that users are able to sign content, a digital signature can be created from the hash, using the user's private key. Either way, the result is then radix-64 encoded to create an ASCII string so as to remove or reduce potential problems with subsequent communication. The openPGP standard ensures that the trasformations and algorithms used to create the signature are indicated within it.

The signature can serve two purposes. If only the hashing step is done, the digest acts as a data integrity check, indicating if the data have been tampered with after creation. If the signing step is carried out, it authenticates the user as the author of the content to readers of the content. In a versioned EHR system, it also acts as a non-repudiation measure, since the signature is stored permanently with the data. To circumvent hacking of the data, public notarisation of the signature can be used. The signature, if present, is generated according to the IETF RFC 2440 (openPGP)[1] standard, following the process shown in FIGURE 10.

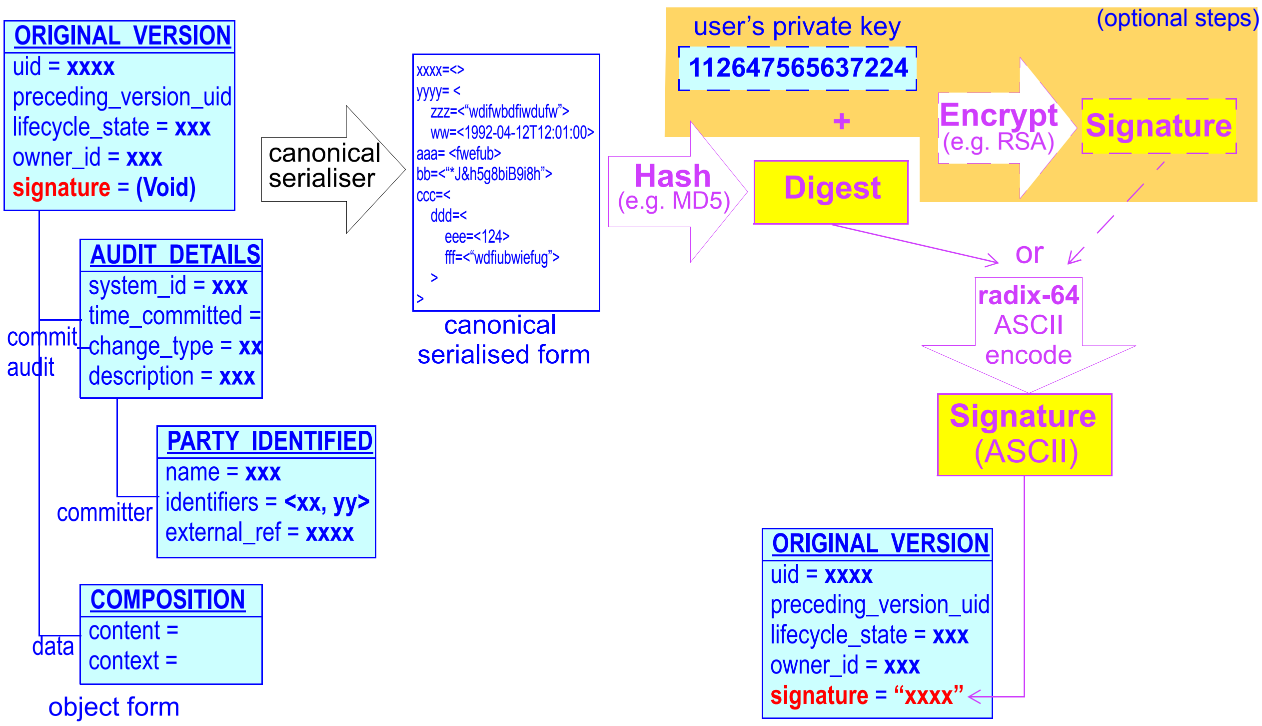


FIGURE 10 Version Signature (using openPGP)

1. See http://www.ietf.org/internet-drafts/draft-ietf-openpgp-rfc2440bis-18.txt

The serialisation process works by the simple rule of serialising the entire Version object (note that the *signature* attribute will be Void at this point) into an agreed XML, dADL or other text format, then applying the subsequent transformations to the serialised data, then writing the digest result back into the *signature* attribute. If the object to be serialised is an IMPORTED_VERSION, the process is the same - all attributes of the object are serialised and then used to generate a signature. The result will be that the IMPORTED_VERSION instance will carry its own signature which signifies the act of importing and making available locally an ORIGINAL_VERSION from another system.

To Be Determined: The exact serialisation is not yet defined by *open*EHR, but dADL might be preferred since it has an unambiguous encoding of object structures, whereas different XML libraries can generate different XML from the same objects.

It should be noted that the signing process here creates a signature of a logical form of the content, not a particular graphical or other directly human interpretable view. Usually the relationship between the data and what is seen on the screen is assumed to be 1:1 in a reliable system. If however the equivalent of a signature of a screen image or other literal form of the data are needed, then the Attestation form of the *commit_audit* is needed. This is described below.

One of the most important uses of signatures in *open*EHR data is likely to be within EHR Extracts, since they can provide an assurance authenticity and integrity of the data to a receiver who has no knowledge of the quality of the processes used in the originating system.

The signing computation has to be performed on the server side of a system, just prior to committal, since one of the data elements included in the signed content is the committal timestamp.

6.2.8 Attestation

The ORIGINAL_VERSION.*attestations* attribute allows attestations to be associated with the data in an original version. Attestations are treated in *open*EHR as a kind of audit with additional attributes, and are described in detail in the common.generic package section of the Common IM. Any number of attestations to be associated with each Version in a Versioned object. Attestations can be added at any time after committal of the content being attested. They can be used as required by enterprise processes or legislation, and indicate by whom 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.

Attestations may be used in different ways as follows.

- *Signing content at committal*: for some reason, the information being committed needs to be digitally signed. It may be that 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, or simply something which the user wants to sign. In this case, ORIGINAL_VERSION.*commit_audit* is of type ATTESTATION rather than AUDIT_DETAILS.
- *Marking content for review and signing*: data entered and committed by a data-entry person e.g. a secretary, transcriptionist or student need to be reviewed and signed by a senior clinician. Similarly to the above case, this will cause ORIGINAL_VERSION.*commit_audit* to be of type ATTESTATION, but in this case, the Attestation will have its *is_pending* flag set True to indicate that attestation is required.
- *Post-committal signing*: data committed with an Attestation in the *is_pending* state is reviewed and signed at a later point in tme by an appropriate member of staff. This action will cause an ATTESTATION to be added to the ORIGINAL_VERSION.*attestations* list.

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`.

When subsequent Versions are added, the existing Attestations can not be assumed to be valid for the new Version, since the nature of an attestation is that it records the witnessing of exactly the content displayed at the time of witnessing.

6.3 Versioning Semantics

6.3.1 Version Lifecycle

Content in Original versions has a lifecycle state associated with it, modelled using the `ORIGINAL_VERSION`.*lifecycle_state* attribute, which is coded from the *open*EHR Terminology "version lifecycle state" group. The possible values are "complete", "incomplete" and "deleted". Usually content will be committed in the "complete" state. However, in some circumstances, e.g. because the author has run out of time or due to an emergency, it may be committed as "incomplete" meaning that it is either 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.

Going from "incomplete" to "complete" 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 incomplete when viewed by a user.

Systems are responsible for implementing checks to find 'old' Versions in "incomplete" state, and bring them to the relevant user's notice, or automatically deleting them or progressing them to "complete" as appropriate.

6.3.2 Logical Deletion

Within the lifecycle described above, deletion of existing top-level content items (i.e. the entire data contents of a Version) is somewhat of a special case in *open*EHR and in EHRs in general. Medico-legal and traceability requirements mean that information cannot be literally removed, since it must always be possible to revert back to a previous state of the record in which the deleted information is intact. Accordingly, information can only ever be logically deleted. This is achieved by the following procedure in the Version container in question:

- create a new Version in the normal way;
- delete its *data* (which will by default be a copy of the data of the previous Version);
- set the *lifecycle_state* value to the code for "deleted"
- commit in the normal way.

Logical deletion can be used for various reasons, including patient direction to remove material, and in the situation where information about a different patient has been incorrectly committed to a record, and has to be removed.

6.3.3 Version Identification

The version identification scheme described here is adapted from the work of Hnìtynka and Plášil [3]. VERSION objects are identified by a *uid* attribute, which is a three-part identifier consisting of the attributes *object_id*, *version_tree_id* and *creating_system_id* (see support.identification package in the Common IM). The first part of the VERSION identifier - the *object_id* attribute - is a copy of the *uid* of the VERSIONED_OBJECT container in which the VERSION was originally created. The second and third parts of the identifier are explained below. FIGURE 11 illustrates the scheme graphically.

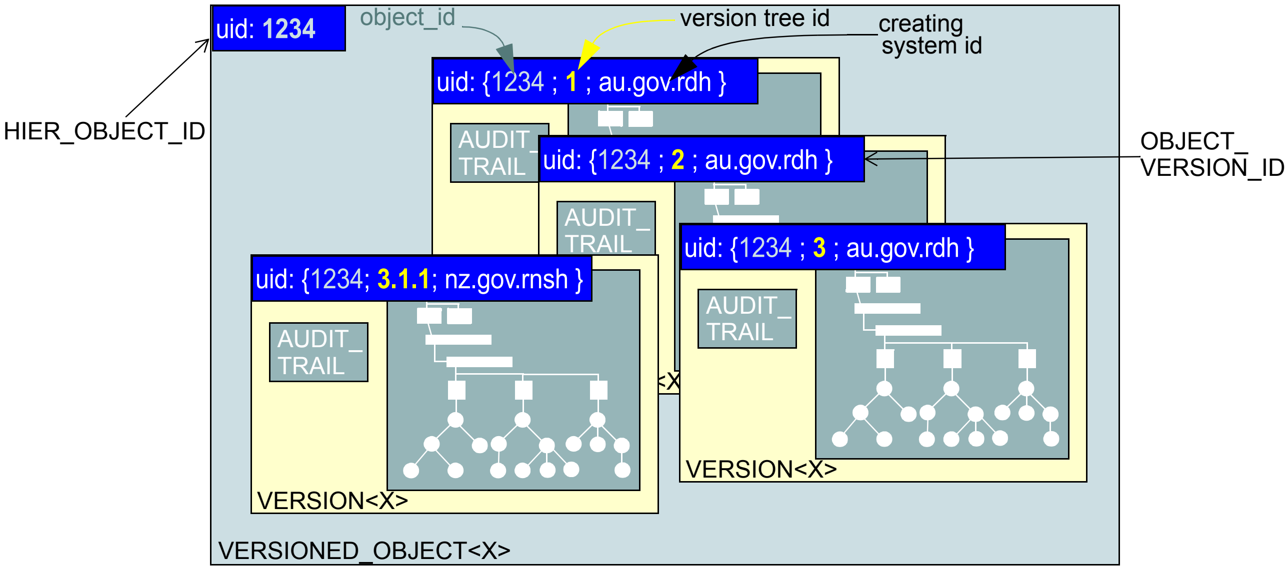


FIGURE 11 Version identification system

Local Versioning

The *version_tree_id* attribute of VERSION.*uid* identifies a version of an item with respect to other versions in the same tree. The requirementsof the identifier 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 such that given a series of ids, the relative positions in the tree can be determined;
- to allow for branches, so that variants of a particular node can be created; e.g. due to translation, or for training purposes.

A suitable scheme satisfying the above requirements for health information is the simplest possible, i.e. a single number representing the version. Version identifiers thus start at 1 and continue 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. Both of these numbers also start at 1. 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) are possible. Inside *open*EHR systems where sharing with other systems does not occur, it is expected that branched versioning will be used rarely; translation is likely to be the only reason (for example if a Portuguese translation of an English language version of a Composition is made).

Distributed Versioning

However, in a distributed environment where copying and subsequent modification can occur, there are more requirements of the version identification scheme, as follows:

- it must be possible for an item to be copied and for local modifications then to be made without causing version clashes;
- it must be possible to send more recent versions from the original system to a target system that has already received earlier versions, and for these versions to be distinguishable from versions in the receiving system, including the previously imported versions - this enables the receiving system to know how and where to commit the received versions;
- 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 version containing changes made to a copied version.

To satisfy these needs, two modifications are made to the identification scheme. The first is the addition of the *creating_system_id* attribute of VERSION.*uid*, representing the system where the version was created. This is a machine processable identifier, such as a reverse internet address or GUID. Whenever a new ORIGINAL_VERSION in a particular VERSIONED_OBJECT (with a particular uid) is created locally, the VERSION.*uid*.*creating_system_id* is set to the identifier of the local system; if the version was imported, *creating_system_id* will already have been set to the identifier of the system of original creation.

The second modification is to require branching version identifiers to be used when local modifications are made to versions copied from elsewhere; 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 without version identifier clashes.

In summary, this scheme uses the tuple {*owner_id*, *version_tree_id*, *creating_system_id*} to globally uniquely identify any *open*EHR VERSION object.

6.4 Semantics in Distributed Systems

6.4.1 Copying

The Copy Operation

In *open*EHR, the smallest unit of copying of content between systems that satisfies traceability requirements is the ORIGINAL_VERSION. In order to copy a OBSERVATION or even an COMPOSITION somewhere else and retain versioning capability, its enclosing ORIGINAL_VERSION object must be sent. When the type of content is a COMPOSITION for example, an ORIGINAL_VERSION<COMPOSITION> object is sent. At the receiving system various steps will occur depending on whether:

- any items for the EHR in question have ever been copied before;
- a copied EHR exists in the destination system for the subject of care, but no copies of the particular item in question have even been made (e.g. it is the first time Family History has been copied);
- an EHR exists, and previous copies have been made for the item in question;
- there is a duplicate EHR for the subject of care (i.e. created by new data entry rather than by automatic copying).

In the first situation, there is not even an EHR (i.e. repository of Versioned objects for the patient in question) in the target system. A new one has to be created. As mentioned in the EHR IM document, the newly created EHR should re-use the EHR id from the source system. This establishes the new EHR as an intentional clone of the source EHR (or more correctly, part of the family of EHRs making up the virtual EHR for that patient).

If it is the first time *any* version of the item logically identified by its ORIGINAL_VERSION.*uid.object_id* (i.e. the *uid* of its original VERSIONED_OBJECT, common to all Versions in the same container) 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.*uid.object_id*. This establishes the newly created VERSIONED_OBJECT as being a logical clone of the one from which the received ORIGINAL_VERSION was copied. If some version of the item had already been received, this step will have already occurred, and the requisite VERSIONED_OBJECT would already exist.

An IMPORTED_VERSION instance is then created, its *item* set to the received ORIGINAL_VERSION, and it is committed in the normal way (i.e. as part of a Contribution). The IMPORTED_VERSION *commit_audit* and *contribution* attributes record the local act of committal. In this operation, the ORIGINAL_VERSION instance is never modified - it remains a faithful copy of its original, no matter how many systems it may be copied through.

Subsequent Local Modifications

In most cases, the received information will remain 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 locally to a copied object, branching numbering is used in the *uid.version_tree_id*, while the local system id is recorded in the *uid.creating_system_id* attribute.

These copying scenarios are illustrated in FIGURE 12. On the left hand-side of the figure, a version container (i.e. an instance of VERSIONED_OBJECT) with *uid*=1 is shown; the first Version has *uid.creating_system_id*="sysA"; *uid.version_tree_id*="1". Further local trunk and branch versions are also shown.

When the first ORIGINAL_VERSION is copied (copy #1) to system B, it is committed as an IMPORTED_VERSION to a VERSIONED_OBJECT which is a clone of the original. Subsequent copies (copy #2 and copy #3) can be made of later versions from system A to system B, with the effect that the version tree can be recreated inside system B (if required; there is of course no obligation to do anything with the received information). Users in system B an also make modifications to the received Version copies; these modifications are shown in grey, as branched versions with *uid.creating_system_id* = sysB. Independently, users in system B will of course be creating other content locally, e.g. as shown on the right-hand side, where a Versioned object with *uid*=2 has been created. Two places are indicated on the diagram where identification clashes could have occurred, but are prevented due to the use of the 3-part unique Version identifier scheme.

Two rules are required to make this system work, as follows:

- branch versions from the original systems that are copied to another system cannot be copied without their corresponding preceding versions on the same branch (if any) and trunk versions also being copied;
- no system should create a new Versioned object (with a new uid) without first determining that it does not already have one with the same uid. This should happen automatically if

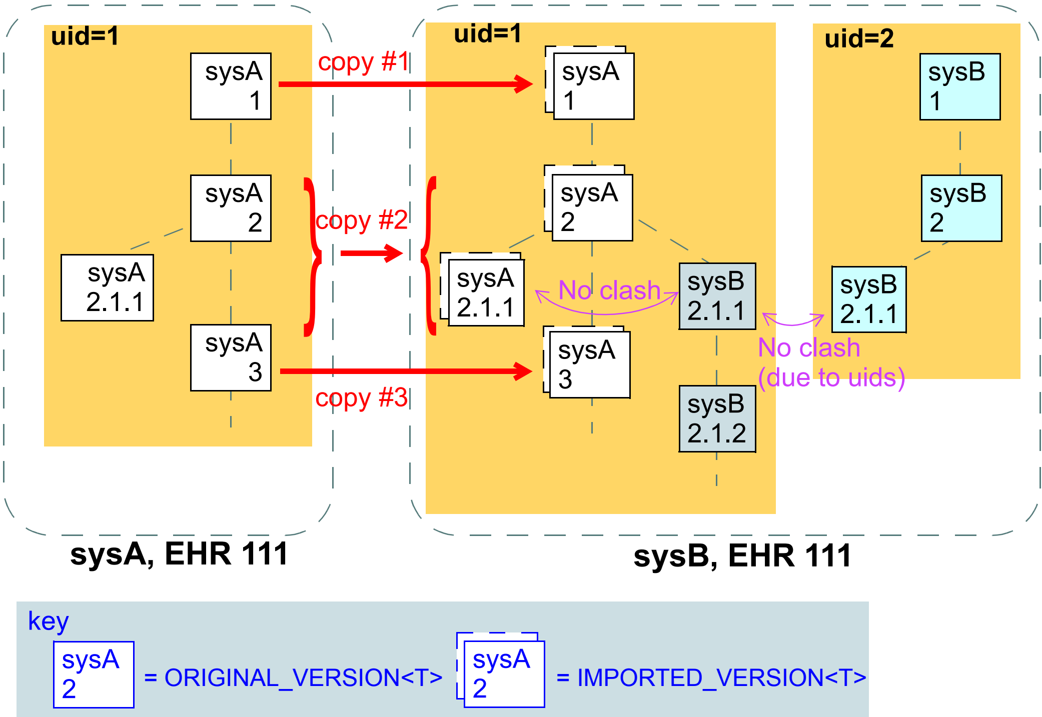


FIGURE 12 Versioning in a distributed environment

GUIDs are being used (and the generating software is reliable); checks may have to be made if ISO Oids are being used.

An important consequence of the way `IMPORTED_VERSION` is modelled is that in the Version containers resulting from copy operations, the commit times always reflect the local (more recent) act of committal, not the original committal of the information to the container where it was created. This ensures that a query for the state of a Version container at earlier commit times correctly returns what information existed at that time in that container, rather than giving the illusion that recently copied Versions were there earlier than the time of local committal (as would occur if the original commit time of the `ORIGINAL_VERSION` object was used for comparison purposes in such queries). Accordingly, such a query over an entire EHR or other versioned information repository always returns the state of the repository available to users at that time, regardless of how many later merges or copies were carried out. This is a key requirement for supporting medico-legal and historical investigations of stored information.

6.4.2 Version Merging

One of the most common operations in distributed versioned environments, particularly in healthcare, is that content created in one system is imported into another system, modifications are created locally there which are then sent back the first system. This information pathway corresponds to scenarios such as the patient being referred from primary care into a hospital, and later being discharged into primary (or other care).

The usual need when the first system receives changes made to the data by the second system is to merge them back into the trunk of the version tree. Logically a 'merge' is the operation of using two versions of the same content to create a third version. How the source versions are used will vary based on the semantics of the information; it could be that the either is simply taken in its totality and the other discarded, or some mixture might be created of the two in a process of editing by the user. In many cases in health, such as where the content is a medication or problem list, the user in the original system will review the received content and create a new trunk version locally using that content, since it will be deemed to be the most accurate available in the clinical computing environment. This scenario is illustrated in FIGURE 13.

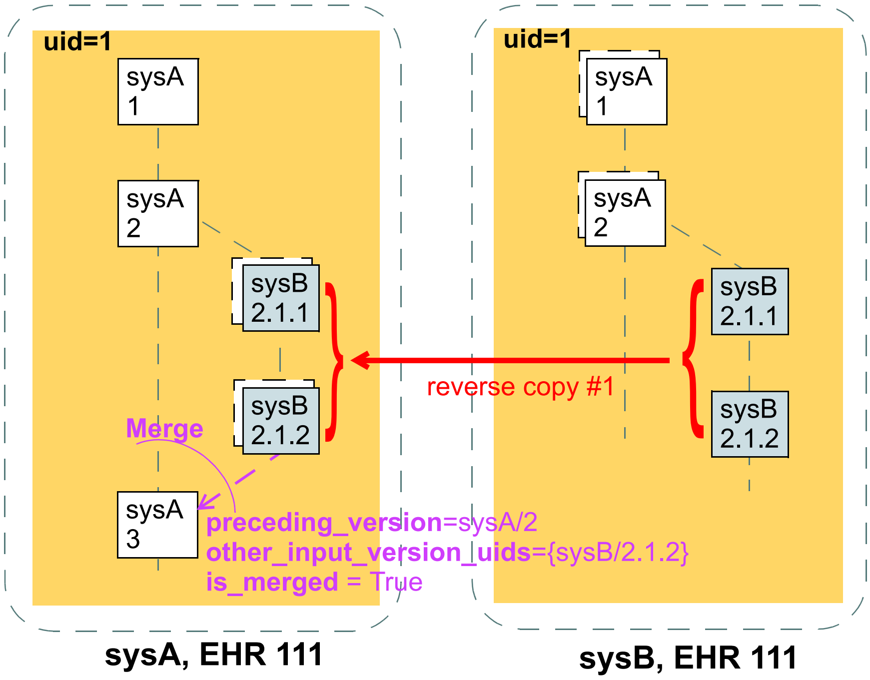


FIGURE 13 Version Merging

In this figure, versions 1 and 2 of the content (e.g. a medication list) from Versioned object with *uid*=1 are copied from system A (e.g. a GP) to system B (e.g. a hospital). In system B, changes are made to version 2, creating a branch (as an instance of `IMPORTED_VERSION<T>`) as required by the rules described above. These changes (modified medication list) are then imported back into system A. The system A user performs a merge operation to create a new trunk version 3, using the sysB/2.1.2 and sysA/2 content; most likely, he simply reviews the two input versions and uses the sysB/2.1.2 content unchanged (the result is that system A now has an up-to-date medication list for the patient, including medications orginally recorded at system A, as well as additions recorded at system B). The new Version is an instance of `ORIGINAL_VERSION<T>`, with its *other_input_version_ids* attribute set to include the `OBJECT_VERSION_ID` representing sysB/2.1.2 (it does not need to include sysA/2, since this is already known in the *preceding_version_uid*).

If in system A a modification had been done to the sysA/2 version, creating sysA/3, in parallel with the system B changes, then a conflict situation is likely when the merge attempt is made. This may need to be resolved by a human user, for whom an automated merge attempt could be presented on the screen as a starting point, much as current source code control tools do today.

6.4.3 Disjoint Merging

An unintended but not uncommon situation is when distinct Version containers are created for the same real-world entity. For example, separate EHRs can be created for the one patient, due to patient identification errors or other procedural or administrative problems. Each record is likely to contain some logically duplicated basic information, as well as information unique to that record, e.g. contributed by different hospital departments. Within the one EHR, unintentionally distinct Version containers might be created for the same logical item, such as the patient’s problem list.

These erroneous situations are eventually detected, and need to be rectified. Logically what is required is to merge the two records (each potentially consisting of numerous Version containers) into one, as shown in FIGURE 14.

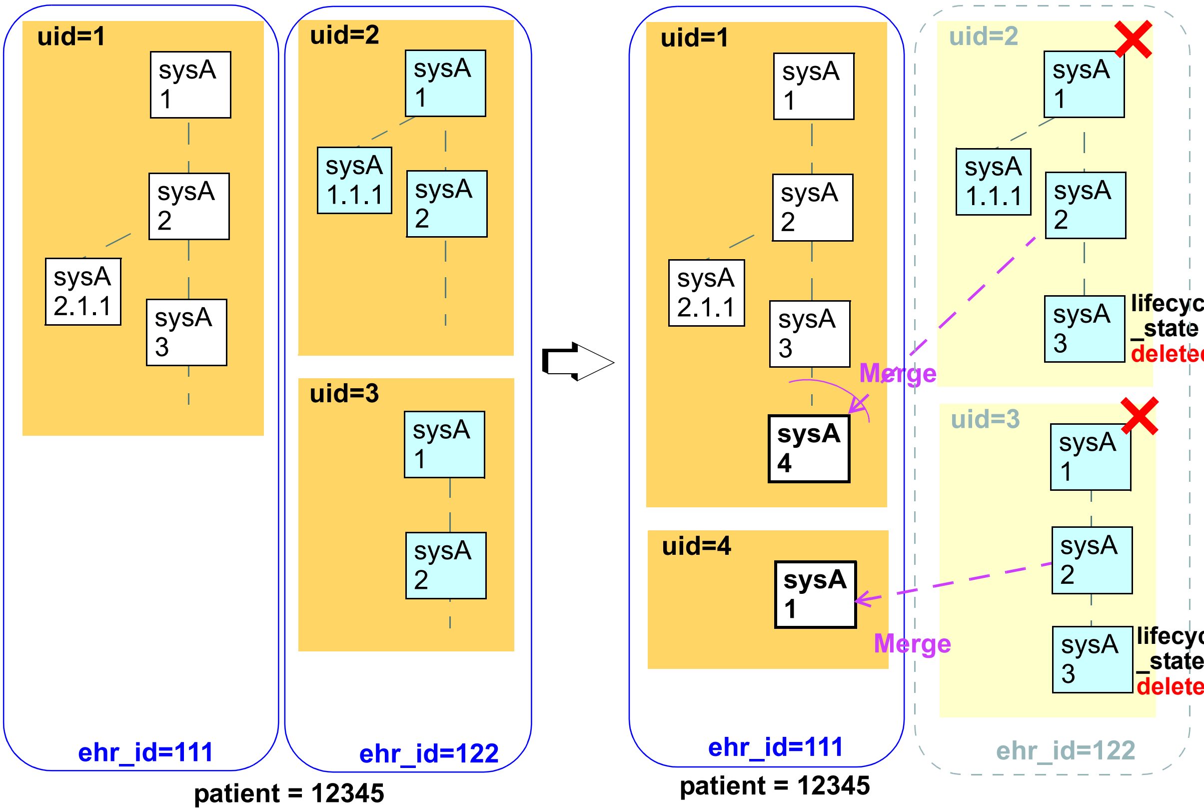


FIGURE 14 Merging of Disjoint Versions

The merge procedure is as follows:

- decide which record is to remain active (for merging purposes, this will be the "target", the other the "source");
- for all Version containers in the source record...
  - if there is a logical equivalent in the target record (for EHRs, there will typically only be equivalents for persistent and possibly administrative Compositions), perform a disjoint merge in the target Version container by:
    - creating a new trunk version in the target Version container;
  - if there is no logical equivalent, do the following:
    - create a new target Version container;
    - create its first trunk Version;
  - in both cases, continue as follows:
    - set the data in the new trunk Version to be a copy of the data from the most recent trunk Version from the source container;
    - set *other_input_version_uids* to include the *uid* of the source Version being merged (this uid will contain the uid of the Version container being logically deleted);
    - for any branches on the most recent trunk Version in the source container, create corresponding branches on the newly created trunk Version in the

target, include the corresponding content and set the *other_input_version_uids* in the target in the same way as above;

* add a new trunk Version to the source container, with the *data* set to Void, and *lifecycle_state* set to `deleted`.

As for copying and merging, an important consequence of this procedure is that the resulting record (i.e. the target of the merge procedure) continues to correctly represent previous states of the repository, regardless of how many recent merges have occurred.

6.4.4 Moving Version Containers

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 re-organisation of EHR data centres. The semantics of a move are different from those of copying: with a move, there is no longer a source instance after the operation; the destination instance becomes the primary instance.

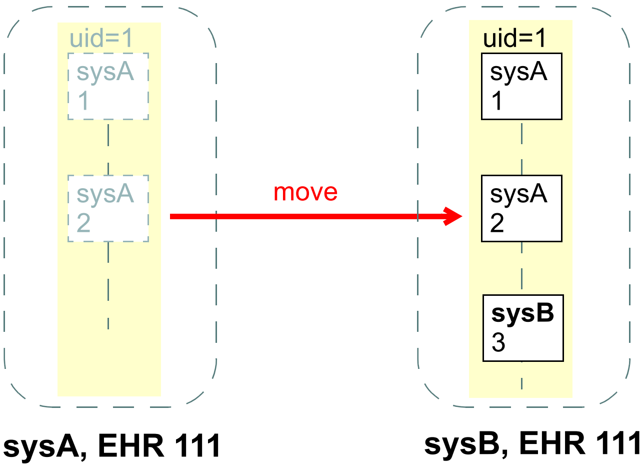


FIGURE 15 A Moved Version Container

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 *uid.creating_system_id* set to the id of the new system. This creates another variation on the version lineage, one in which the *uid.creating_system_id* value can change in the trunk line, as shown in FIGURE 15.

6.5 Class Descriptions

6.5.1 VERSIONED_OBJECT Class

CLASS	**VERSIONED_OBJECT<T>**	
Purpose	Version control abstraction, defining semantics for versioning one complex object.	
Attributes	**Signature**	**Meaning**

CLASS	VERSIONED_OBJECT<T>	
1..1	**uid**: `HIER_OBJECT_ID`	Unique identifier of this version container. This id will be the same in all instances of the same container in a distributed environment, meaning that it can be understood as the uid of the "virtual version tree".
1..1	**owner_id**: `OBJECT_REF`	Reference to object to which this version container belongs, e.g. the id of the containing EHR or other relevant owning entity.
1..1	**time_created**: `DV_DATE_TIME`	Time of initial creation of this versioned object.
Functions	**Signature**	**Meaning**
1..1	**all_versions**: `List <VERSION<T>>`	Return a list of all versions in this object.
1..1	**all_version_ids**: `List <OBJECT_VERSION_ID>`	Return a list of ids of all versions in this object.
1..1	**version_count**: `Integer`	Return the total number of versions in this object
	has_version_id (a_ver_id: `OBJECT_VERSION_ID`): `Boolean` **require** a_ver_id /= Void	True if a version with an_id exists.
	is_original_version (a_ver_id: `OBJECT_VERSION_ID`): `Boolean` **require** a_ver_id /= Void **and** has_version_id(a_ver_id)	True if version with an_id is an `ORIGINAL_VERSION`.
	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 (a_ver_id: `OBJECT_VERSION_ID`): `VERSION<T>` **require** has_version_id(a_ver_id)	Return the version with id 'an_id'.

CLASS	VERSIONED_OBJECT<T>	
	version_at_time (a_time: DV_DATE_TIME): VERSION<T> **require** has_version_at_time(a_time)	Return the version for time ‘a_time’.
	latest_version: VERSION<T>	Return the most recently added version (i.e. on trunk or any branch).
	latest_trunk_version: VERSION<T>	Return the most recently added trunk version.
1..1	**trunk_lifecycle_state**: DV_CODED_TEXT ***ensure*** Result = latest_trunk_version.lifecycle_state	Return the lifecycle state from the latest trunk version. Useful for determining if the version container is logically deleted.
1..1	**revision_history**: REVISION_HISTORY	History of all audits and attestations in this versioned repository.
	commit _original_version (a_contribution: OBJECT_REF; a_new_version_uid, a_preceding_version_uid: OBJECT_VERSION_ID; an_audit: AUDIT_DETAILS; a_lifecycle_state: DV_CODED_TEXT; a_data: T; signing_key: String) **require** ***Contribution_valid***: a_contribution /= Void ***New_version_valid***: a_new_version_uid /= Void ***Preceding_version_uid_valid***: all_version_ids.has(a_preceding_version_uid) **or else** version_count = 0 ***audit_valid***: an_audit /= Void ***data_valid***: a_version_data /= Void ***lifecycle_state_valid***: a_lifecycle_state /= Void	Add a new original version.

CLASS	VERSIONED_OBJECT<T>	
	commit _original_merged_version (a_contribution: OBJECT_REF; a_new_version_uid, a_preceding_version_uid: OBJECT_VERSION_ID; an_audit: AUDIT_DETAILS; a_lifecycle_state: DV_CODED_TEXT; a_data: T; an_other_input_uids: Set<OBJECT_VERSION_ID>; signing_key: String) **require** ***Contribution_valid***: a_contribution /= Void ***New_version_valid***: a_new_version_uid /= Void ***Preceding_version_id_valid***: all_version_ids.has(a_preceding_version_uid) **or else** version_count = 0 ***audit_valid***: an_audit /= Void ***data_valid***: a_version_data /= Void ***lifecycle_state_valid***: a_lifecycle_state /= Void ***Merge_input_ids_valid***: an_other_input_uids /= Void	Add a new original merged version. This commit function adds a parameter containing the ids of other versions merged into the current one.
	commit _imported_version (a_contribution: OBJECT_REF; an_audit: AUDIT_DETAILS; a_version: ORIGINAL_VERSION<T>) **require** ***Contribution_valid***: a_contribution /= Void ***audit_valid***: an_audit /= Void ***Version_valid***: a_version /= Void	Add a new imported version. Details of version id etc come from the ORIGINAL_VERSION being committed.
	commit_attestation (an_attestation: ATTESTATION; a_ver_id: OBJECT_VERSION_ID; signing_key: String) **require** ***Attestation_valid***: an_attestation /= Void ***Version_id_valid***: has_version_id(a_ver_id) **and** is_original_version(a_ver_id)	Add a new attestation to a specified original version. Attestations can only be added to Original versions.

CLASS	VERSIONED_OBJECT<T>
Invariant	***uid_valid***: uid /= Void ***owner_id_valid:*** owner_id /= Void ***time_created_valid***: time_created /= Void ***version_count_valid***: version_count >= 0 ***all_version_ids_valid***: all_version_ids /= Void **and then** all_version_ids.count = version_count ***all_versions_valid***: all_versions /= Void **and then** all_versions.count = version_count ***latest_version_valid***: version_count > 0 **implies** latest_version /= Void ***revision_history_valid***: revision_history /= Void

6.5.2 VERSION Class

CLASS	*VERSION<T> (abstract)*	
Purpose	Abstract model of one Version within a Version container, containing data, commit audit trail, and the identifier of its Contribution.	
Abstract	**Signature**	**Meaning**
1..1	***uid***: OBJECT_VERSION_ID	Unique identifier of this version, containing *owner_id*, *version_tree_id* and *creating_system_id*.
0..1	***preceding_version_uid***: OBJECT_VERSION_ID	Unique identifier of the version of which this version is a modification; Void if this is the first version.
0..1	***data***: T	Original content of this Version.
1..1	***lifecycle_state***: DV_CODED_TEXT	Lifecycle state of this version; coded by *open*EHR vocabulary “version lifecycle state”.
Attributes	**Signature**	**Meaning**
1..1	**commit_audit**: AUDIT_DETAILS	Audit trail corresponding to the committal of this version to the VERSIONED_OBJECT.
1..1	**contribution**: OBJECT_REF	Contribution in which this version was added.
0..1	**signature**: String	OpenPGP digital signature or digest of content committed in this Version.
Functions	**Signature**	**Meaning**
1..1	**owner_id**: HIER_OBJECT_ID	Unique identifier of the owning VERSIONED_OBJECT.
1..1	**is_branch**: Boolean	True if this Version represents a branch. Derived from *uid* attribute.

CLASS	VERSION<T> (abstract)	
1..1	**canonical_form**: String	Canonical form of Version object, created by serialising all attributes except *signature*.
Invariant	***Uid_valid***: uid /= Void ***Owner_id_valid***: owner_id /= Void **and then** owner_id.value.is_equal(uid.object_id.value) ***Commit_audit_valid***: commit_audit /= Void ***Contribution_valid***: contribution /= Void and contribution.type.is_equal("CONTRIBUTION") ***Preceding_version_uid_validity***: uid.version_tree_id.is_first **xor** preceding_version_uid /= Void ***Lifecycle_state_valid***: lifecycle_state /= Void **and then** terminology(Term_id_openehr).has_code_for_group_id(Group_id_version_lifecycle_state, lifecycle_state.defining_code)	

6.5.3 ORIGINAL_VERSION Class

CLASS	ORIGINAL_VERSION<T>	
Purpose	A Version containing locally created content and optional attestations.	
Attributes	Signature	Meaning
1..1 (effected)	**uid**: OBJECT_VERSION_ID	Stored version of inheritance precursor.
0..1 (effected)	**preceding_version_uid**: OBJECT_VERSION_ID	Stored version of inheritance precursor.
0..1	**other_input_version_uids**: Set<OBJECT_VERSION_ID>	Identifiers of other versions whose content was merged into this version, if any.
0..1 (effected)	**data**: T	The data being versioned. If not present, this corresponds to logical deletion.
0..1	**attestations**: List <ATTESTATION>	Set of attestations relating to this version.
1..1 (effected)	**lifecycle_state**: DV_CODED_TEXT	Lifecycle state of the content item in this version.
Functions	Signature	Meaning
	is_merged: Boolean	True if this Version was created from more than just the preceding (checked out) version.

CLASS	ORIGINAL_VERSION<T>
Invariant	***Attestations_valid***: attestations /= Void **implies not** attestations.is_empty ***Is_merged_validity***: other_input_version_ids = Void **xor** is_merged ***Other_input_version_uids_valid***: other_input_version_uids /= Void **implies not** other_input_version_uids.is_empty

6.5.4 IMPORTED_VERSION Class

CLASS	IMPORTED_VERSION<T>	
Purpose	Versions whose content is an ORIGINAL_VERSION copied from another location; this class inherits *commit_audit* and *contribution* from VERSION<T>, providing imported versions with their own audit trail and Contribution, distinct from those of the imported ORIGINAL_VERSION.	
Inherit	VERSION<T>.	
Attributes	**Signature**	**Meaning**
1..1	**item**: ORIGINAL_VERSION<T>	The ORIGINAL_VERSION object that was imported.
Functions	**Signature**	**Meaning**
(effected)	**uid**: OBJECT_VERSION_ID ***ensure*** Result = item.uid	Computed version of inheritance precursor, derived as *item.uid*.
(effected)	***data***: T	Data of wrapped ORIGINAL_VERSION.
(effected)	**preceding_version_uid**: OBJECT_VERSION_ID ***ensure*** Result = item.preceding_version_uid	Computed version of inheritance precursor, derived as *item.preceding_version_uid*.
(effected)	**lifecycle_state**: DV_CODED_TEXT	Lifecycle state of the content item in wrapped ORIGINAL_VERSION, derived as *item.lifecycle_state*.
Invariant	***Item_valid***: item /= Void	

6.5.5 CONTRIBUTION Class

CLASS	CONTRIBUTION
Purpose	Documents a contribution of one or more versions added to a change-controlled repository.

CLASS	CONTRIBUTION	
Attributes	**Signature**	**Meaning**
1..1	**uid**: HIER_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_valid***: uid /= Void ***audit_valid***: audit /= Void ***Versions_valid***: versions /= Void **and then not** versions.is_empty ***Description_valid***: audit.description /= Void	

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 16.

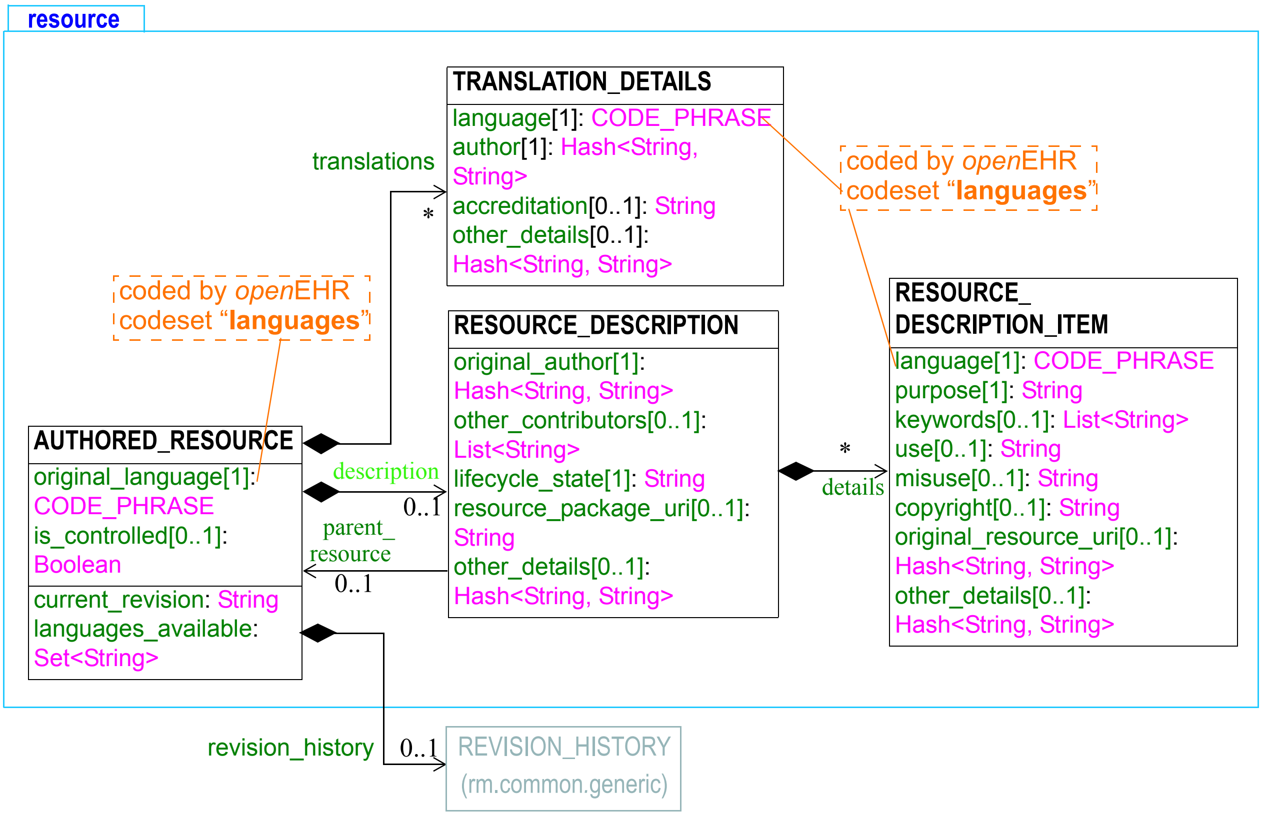


FIGURE 16 openehr.rm.common.resource Package

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.

The AUTHORED_RESOURCE.*description* attribute is optional, allowing for resources with no meta-data at all, e.g. resources in a partial state of construction. The *translations* attribute may still be required, since there may be other parts of the resource object (specified by a class into which AUTHORED_RESOURCE is inherited) that are language-dependent.

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. The *original_language* does not appear in this list.

CLASS	*AUTHORED_RESOURCE (abstract)*	
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).
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**
1..1	**current_revision**: String ***ensure*** Result = revision_history. most_recent_version	Most recent revision in *revision_history* if *is_controlled* else "(uncontrolled)".
1..1	**languages_available**: Set<String>	Total list of languages available in this resource, derived from *original_language* and *translations*.
Invariant	***Original_language_valid***: original_language /= void **and then** code_set(Code_set_id_languages).has_code(original_language.as_string) ***Languages_available_valid***: languages_available /= Void **and then** languages_available.has(original_language) ***Revision_history_valid***: is_controlled **xor** revision_history = Void ***Current_revision_valid***: (current_revision /= Void **and not** is_controlled) **implies** current_revision.is_equal("(uncontrolled)") ***Translations_valid***: translations /= Void **implies** (**not** translations.is_empty **and not** translations.has(orginal_language.code_string)) ***Description_valid***: translations /= Void **implies** (description.details.for_all(d | translations.has_key(d.language.code_string)))	

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

CLASS	TRANSLATION_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(Code_set_id_languages).has_code(language) ***Author_valid***: author /= Void	

7.2.3 RESOURCE_DESCRIPTION Class

CLASS	RESOURCE_DESCRIPTION	
Purpose	Defines the descriptive meta-data of a resource.	
Attributes	Signature	Meaning
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**: `Hash <RESOURCE_DESCRIPTION_ITEM, String>`	Details of all parts of resource description that are natural language-dependent, keyed by language code.
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.
0..1	**parent_resource**: `AUTHORED_RESOURCE`	Reference to owning resource.

CLASS	RESOURCE_DESCRIPTION
Invariant	***Original_author_valid***: original_author /= Void **and then not** original_author.is_empty ***Lifecycle_state_valid***: lifecycle_state /= Void **and then not** lifecycle_state.is_empty ***Details_valid***: details /= Void **and then not** details.is_empty ***Language_valid***: parent_resource /= Void **implies** details.for_all (d | parent_resource.languages_available.has(d.language.code_string)) ***Parent_resource_valid***: parent_resource /= Void **implies** 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(Code_set_id_languages).has_code(language) ***purpose_valid***: 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