Knowledge Artefact Identification

Issuer: openEHR Specification Program		
Revision: 0.7.1	*Pages:* 41	*Date of issue:* 09 May 2014
Status: DEVELOPMENT		

Keywords: EHR, health records, repository, governance

Amendment Record

Issue	Details	Who	Completed
0.7.1	Simplify development state in lifecycle; merge ‘initial’ and rename ‘draft’.	S Garde I McNicoll	09 May 2014
0.7.0	Rewrite referencing section; update ‘problem description; further grammar improvements.	T Beale	20 Jun 2013
0.6.5	Remove errors and ambiguities to do with explanation of human readable identifier; improve nomenclature; rewrite grammar.	T Beale	15 Jun 2013
0.6.0	Major update based on CKM clinical group analysis, and feedback from the CIMI and *open*EHR communities.	S Garde H Leslie I McNicoll T Beale	21 Apr 2013
0.2.0	Refinements to do with template identification. Review from Medical Centrum Alkmaar (Netherlands).	T Beale M van der Meer	01 Feb 2010
0.1.0	Initial Writing.	T Beale	09 Jul 2009

Trademarks

“Microsoft” and “.Net” are registered trademarks of the Microsoft Corporation.

“Java” is a registered trademark of Sun Microsystems.

“Linux” is a registered trademark of Linus Torvalds.

Acknowledgements

The work reported in this document was funded by:

- University College London, Centre for Health Informatics and Multi-professional Education (CHIME);
- Ocean Informatics.

1 Introduction

1.1 Purpose

The purpose of this document is to describe an identification system for health informatics knowledge artefacts, including archetype, template and terminology subsets. This includes such artefacts created by organisations such as the *open*EHR Foundation, standards bodies and clinical modelling initiatives.

The semantics covered include:

- formal human-readable and machine identifiers;
- versioning;
- lifecycle management and states;
- referencing artefacts from elsewhere;
- deal with transfer and forking;
- supporting integrity and non-repudiation.

Unless otherwise stated, in this document, the term 'artefact' refers specifically to these artefact types.

1.2 Related Documents

This document is part of a framework of documents for which the core document is the following:

- Distributed Development and Governance Model.

1.3 Status

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

The latest version of this document can be found in PDF format at http://www.openehr.org/releases/trunk/architecture/am/knowledge_id_system.pdf. New versions are announced on openehr-announce@openehr.org.

2 Introduction

2.1 The Environment

This specification is designed to address the need for reliable identification and referencing of complex knowledge artefacts within a distributed authoring and consumption environment. The figure below establishes the key concepts and nomenclature assumed by this specification. The focus of interest is 'artefacts', including archetypes, templates (of the archetype variety), terminology subsets/ref-sets, query sets, and potentially things such as computable guidelines.

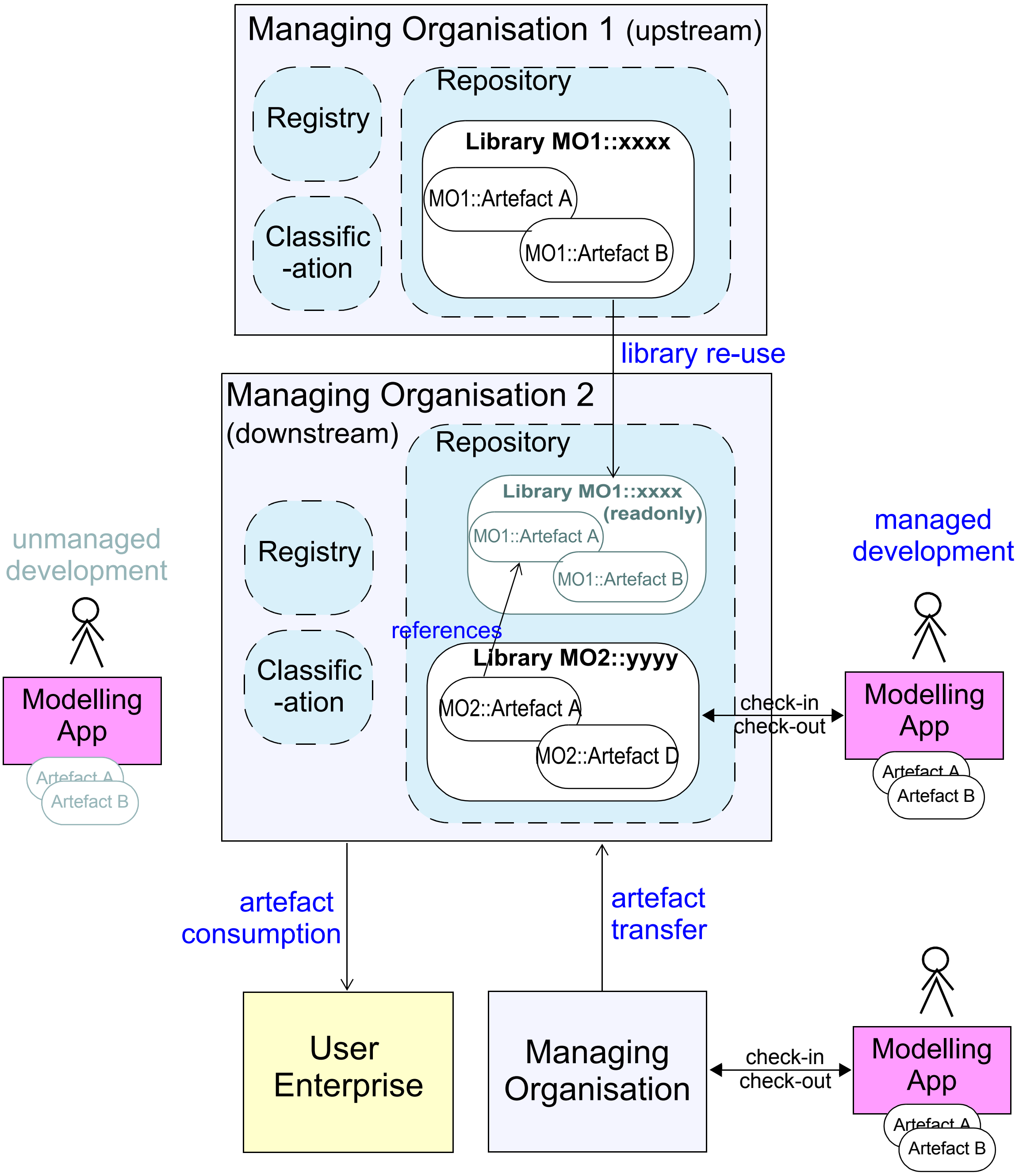


FIGURE 1 Distributed Development Environment

Artefacts are assumed to be produced by tools, either in an unmanaged way, or in a situation in which users are connected to an artefact Managing Organisation (MO). Such an organisation is assumed to have a Repository (which stores and manages artefacts), and potentially a Registry (in which meta-data about artefacts is stored) and Classification (a semantic index on Artefacts, typically achieved via the use of one or more ontologies). A Managing Organisation could be international, country level, or be owned by a company or other organisation.

Most MOs will tend to develop artefacts based on those published by 'higher-level' (i.e. national, international) MOs. To enable this, a logical ability to re-use specified releases of artefacts from 'upstream' MOs by 'downstream' MOs is assumed. This usually implies some kind of virtual inclusion (e.g. from one web-visible repository to another), or it may be implemented by copying and marking as read-only the received artefacts. Regardless of the particular implementation, the 'logical contents' of a repository is the totality of localy managed artefacts, plus all virtually referenced artefact libraries. This is necessary to enable most compiler-like tools to function normally.

It is assumed that Artefacts can also move between MOs for purposes of transfer, or due to 'forking' (i.e. splitting of a line of development, as with software). Artefacts are published in some form and consumer by User Enterprises which deploy the artefacts in some technical infrastructure.

Artefacts are ultimately consumed by User Enterprises, normally in a validated and compiled form.

2.2 The Problem

The problem specifically addressed by this specification is that of identification and referencing of knowledge artefacts. The notion of 'identification' for such artefacts incorporates a number of key requirements. The kinds of models in scope include archetypes[1], templates[2], terminology subsets[3], clinical guidelines, query sets and other non-atomic domain level definitions of content, rules, workflows and other semantics. The common aspects of artefacts within this scope is that they are 'outside the software', and that they are independent of specific implementation technologies. Examples include:

- an archetype for 'blood gases';
- a template for 'discharge summary';
- a SNOMED CT subset for 'parasitic infection'.

Out of scope are the atomic 'concepts' and 'categories' commonly found in terminologies (e.g.: ICD10, SNOMED CT, LOINC) and ontologies (e.g. the BFO ontologies such as OGMS, FMA, IAO etc).

Extensive experience with such artefacts in the health domain has shown that while there are many similarities to software artefact identification, there are sufficient differences to warrant an explicit scheme. The health domain is the primary domain of experience assumed here, but the principles are applicable to any domain.

The key requirements addressed here are as follows:

- identify and distinguish versions, variants and releases of 'source' artefacts within and from *authoring* environments;

[1] http://www.openehr.org/releases/trunk/architecture/am/aom1.5.pdf

[2] http://www.openehr.org/releases/trunk/architecture/am/tom1.5.pdf

[3] various descriptions at http://ihtsdo.org

- define rules for expressing and resolving *references* between source artefacts, including version variants;
- define rules for identification of *compiled* / *operational* artefacts;
- define rules for *evolving identifiers* (including version) of artefacts over time, based on a ‘standard’ lifecycle for artefacts;
- define rules for identification when artefacts are retired, moved or ‘forked’.

2.3 Human-readable and Machine Identifiers

There are two general approaches to identification. The first is the one used in software and ontology development: human-readable identifiers, denoted in this specification as HRIDs. Under this approach, identifiers name an artefact (e.g. a class in object-oriented software, category in an ontology) and can be used as references to connect similar artefacts in a hierarchy (e.g. according to the inheritance relationship). The second is the use of meaningless machine identifiers (more properly denoted ‘machine-readable’ or ‘machine-resolvable’ identifiers) such as GUIDs and ISO OIDs with accompanying de-referencing mechanisms. The two approaches are not mutually exclusive, nor are they equivalent.

A human-readable identification scheme supports the notion of a specialsiation / subsumption hierarchy of artefacts (‘inheritance’ in object programming), multi-dimensional concept spaces, flexible versioning, and formally reflects the artefact authors' and users' understanding of the concept space being modelled. Human-readable identification supports many types of computational processing. A typical software HRID is the class name `FastSortedList`. Within the software world, HRIDs are used for both source artefacts and built components such as libraries and executables, although the details of the respective types of identifier may differ.

One crucial feature of most human-readable identifiers is that they *may change after initial assignment*, for reasons of change of purpose, improved understanding of need, or external requirements change. These kinds of changes are normally limited to the early development (typically pre v1.0 phase) period in order to enable stability later on.

Machine identifiers on the other hand are not human-readable, typically do not directly support versioning (unless specifically designed to do so, usually via the use of tuples of atomic identifiers), but do enable various useful kinds of computation. They require mapping to convert to human-readable identifiers. Unlike human-readable identifiers, machine identifiers do not normally change once assigned.

One key question when using machine identifiers is: what do the identify? A logical artefact, which may exist in several minor and major versions? Each minor version? Each textually different variant that is committed to a repository? For each of these, a scheme has to be devised that correctly identifies the thing to be tracked.

It is possible to define an identification scheme in which either or both human-readable and machine identifiers are used. In schemes where machine identification alone is used, all human artefact 'identification' is relegated to meta-data description, such as names, purpose, and so on. One problem with such schemes is that meta-data characteristics are informal, and therefore can clash – preventing any formalisation of the ontological space occupied by the artefacts. Discovery of overlaps and in fact any comparative feature of artefacts cannot be easily formalised, and therefore cannot be made properly computable.

The approach assumed here is to use both types of identifier in the following way:

- a Guid is assigned to a knowledge artefact when it is created. It does not change, no matter what changes are made to the definition of the artefact. This enables authoring and model repository tools to track artefacts as they are modified over time.
- one or more namespaced HRIDs for an artefact can be *computed* from various properties of the artefact. Which properties will depend on the type of artefact.
- the last committed ‘build’ of an artefact (i.e. most recent version containing a change, no matter how small) can be identified in two ways:
    - using a ‘build’ number that is part of the version identification of the artefact;
    - via a hash on a canonical serialisation of the artefact.

This is a departure from the common situation where no machine identifier is assigned, and the artefact HRID is a static string, rather like a source file filename.

2.4 Meta-data

A solution for identification that includes human readable (formal) identifiers unavoidably implicates the ‘meta-data’ of the identified artefacts, since such identifiers are normally created from smaller items such as ‘reference model class’, ‘version’, ‘namespace’ and so on. However, some items of meta-data are not appropriate for inclusion in an artefact, and would be created in the Registry instead. A general rule is that this applies to any item of information that may change without affecting the semantics of the artefact, and whose change should not require revision of the artefact itself. Examples of such information: ontological classification(s); ‘ownership’ status.

This specification assumes that an artefact management environment includes such a registry, and that some items of meta-data can be stored outside the artefacts themselves.

3 Source Artefact Identification

3.1 Overview

The basis for identifying *source* (i.e. authored) artefacts is to define a number of separate logically identifying *properties*, as well as a machine identifier. One or more human-readable identifier(s) can be generated from the non-uid identifying properties. For archetypes and templates, the relevant properties are defined on the `ARCHETYPE_HRID` class from the *open*EHR Archetype Object Model, shown in blue on the following figure. For completeness, the `AUTHORED_RESOURCE` classes inherited into `ARCHETYPE` are shown, as they supply the `lifecycle_state` property, as well as all other descriptive meta-data.

For other types of artefacts the detailed model will differ, but the principles are the same.

Three distinct groups of properties shown in the `ARCHETYPE_HRID` class in FIGURE 2 that underpin the identification scheme described here, as follows:

- `namespace` provides a way of distinguishing logical identifiers created by different organisations that would otherwise compete in a single semantic identifier space;
- `rm_publisher`, `rm_closure`, `rm_class`, `concept_id` form the basis of the main part of a human-readable identifier, e.g. `openEHR-EHR-OBSERVATION.bp_measurement`;
- properties supporting versioning:
    - `release_version`, expressing a 3-part version identifier, e.g. ‘1.3.0’;
    - `commit_number`, incremented at every commit, supporting non-release version ids, such as ‘1.3.0-rc28’ and ‘1.3.0+u96’, where the commit number is 28 and 96 respectively;
    - `description.lifecycle_state`, expressing the development state of the artefact, and used to derive the ‘rc’ (release candidate) and ‘u’ (development) parts of non-release version ids.

Functions such as `interface_id, physical_id` and `version_id` are defined to return respectively the ‘interface’ and ‘physical’ archetype HRIDs (described below) as strings, and the full version string (computed from `release_version`, `commit_number` and `description.lifecycle_state`). The functions `major_version`, `minor_version` and `patch_version` extract the various parts of the 3-part `release_version` property.

The `uid` property provides the machine identifier, and is assumed to be a Guid.

Both the `uid` and `namespace` properties are optional for legacy reasons, since most existing archetypes have neither. The interpretation of an artefact without these identifiers in this specification is that it is *unmanaged*, i.e. it has no recognised owner organisation. During a period of changeover to the identifiers specified here, there will clearly be artefacts that are in fact managed, and which need to have the `uid` and `namespace` properties assigned. This will obviously take some time, as it requires support from the tooling ecosystem.

Different types of human-readable identifiers are used for archetypes, templates and terminology subsets. The following sections describe the formal details of this identification scheme, and how it supports referencing between artefacts.

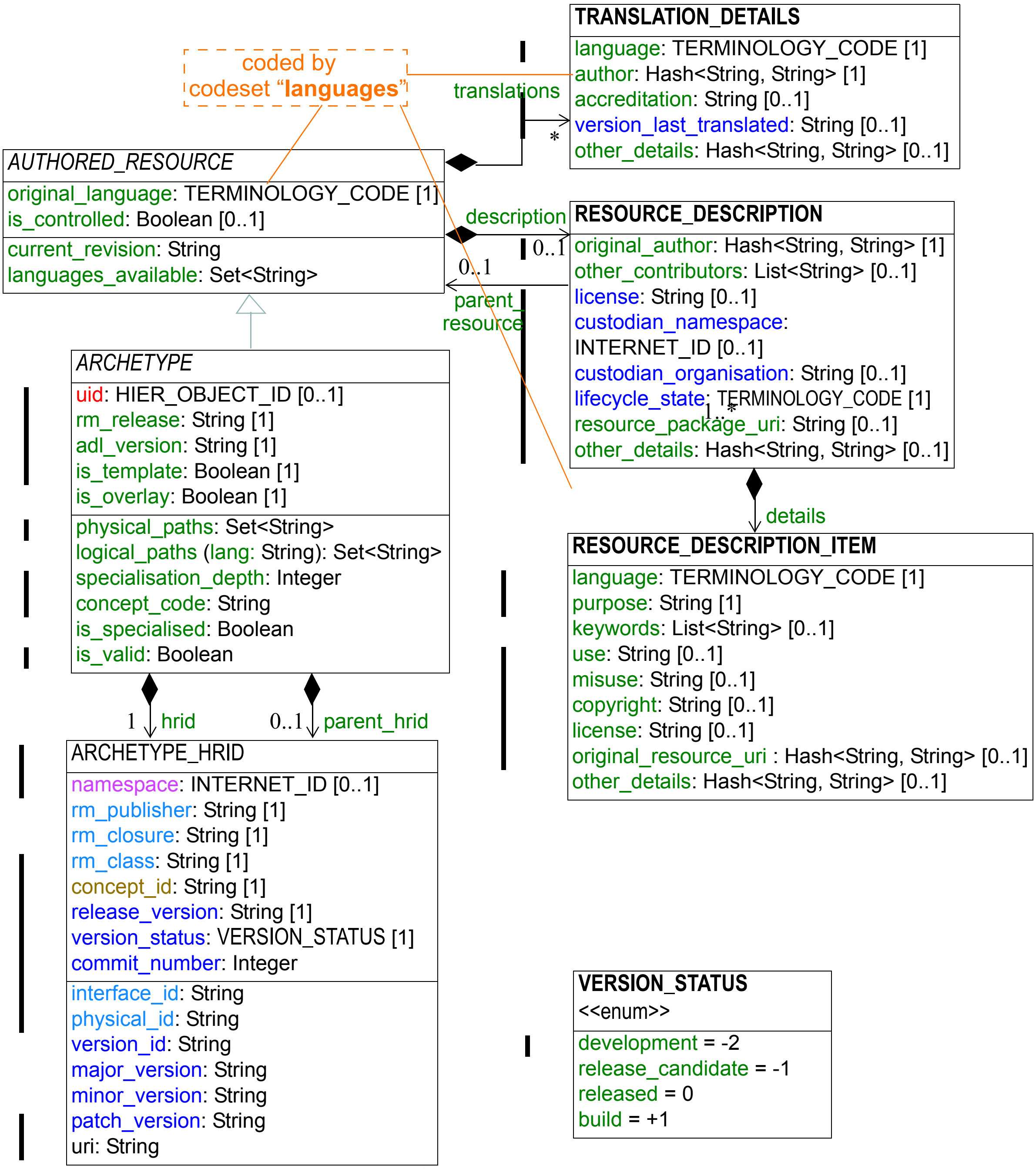

coded by
codeset "languages"
translations
*
TRANSLATION_DETAILS
language: TERMINOLOGY_CODE [1]
author: Hash<String, String> [1]
accreditation: String [0..1]
version_last_translated: String [0..1]
other_details: Hash<String, String> [0..1]
AUTHORED_RESOURCE
original_language: TERMINOLOGY_CODE [1]
is_controlled: Boolean [0..1]
current_revision: String
languages_available: Set<String>
description
0..1
0..1
parent_
resource
RESOURCE_DESCRIPTION
original_author: Hash<String, String> [1]
other_contributors: List<String> [0..1]
license: String [0..1]
custodian_namespace:
INTERNET_ID [0..1]
custodian_organisation: String [0..1]
lifecycle_state: TERMINOLOGY_CODE [1]
1..*
resource_package_uri: String [0..1]
other_details: Hash<String, String> [0..1]
details
RESOURCE_DESCRIPTION_ITEM
language: TERMINOLOGY_CODE [1]
purpose: String [1]
keywords: List<String> [0..1]
use: String [0..1]
misuse: String [0..1]
copyright: String [0..1]
license: String [0..1]
original_resource_uri : Hash<String, String> [0..1]
other_details: Hash<String, String> [0..1]
ARCHETYPE
uid: HIER_OBJECT_ID [0..1]
rm_release: String [1]
adl_version: String [1]
is_template: Boolean [1]
is_overlay: Boolean [1]
physical_paths: Set<String>
logical_paths (lang: String): Set<String>
specialisation_depth: Integer
concept_code: String
is_specialised: Boolean
is_valid: Boolean
1
hrid
0..1
parent_hrid
ARCHETYPE_HRID
namespace: INTERNET_ID [0..1]
rm_publisher: String [1]
rm_closure: String [1]
rm_class: String [1]
concept_id: String [1]
release_version: String [1]
version_status: VERSION_STATUS [1]
commit_number: Integer
interface_id: String
physical_id: String
version_id: String
major_version: String
minor_version: String
patch_version: String
uri: String
VERSION_STATUS
<<enum>>
development = -2
release_candidate = -1
released = 0
build = +1

FIGURE 2 AOM classes containing meta-data

3.2 Formal Model

This section defines a formal grammar for human-readable indentifiers of knowledge artefacts. As described above, more than one human-readable identifier can be constructed from the identifying properties shown in FIGURE 2 above. The grammar is shown in green below.

The highest level distinction is between *managed* and *unmanaged* artefacts, with managed status being indicated by the prepending of a namespace to what can be termed a 'local' HRID (i.e. local to a given namespace context).

```
    artefact_hrid:             namespaced_hrid | local_hrid
```

```
namespaced_hrid:         namespace '::' local_hrid
local_hrid:              hrid_root '.v' version_id
namespace:               V_REVERSE_DOMAIN_NAME
V_REVERSE_DOMAIN_NAME:   See IETF RFCs 1035, 123, and 2181.
```

The `namespace` is the publisher organisation reverse domain name. Reverse domain names are used in order to aid lexical sorting of identifiers and also tools that build directory structures based on reverse domain name segments. All managed artefacts, including archetypes and templates should include a namespace. Any archetype or template carriying an identifier without a namespace is assumed to be an unmanaged artefact.

Examples:

`org.openehr`	EHR archetypes library at *open*EHR.org
`uk.nhs`	UK National Health Service
`edu.nci`	US National Cancer Institute

3.2.1 Human-readable Identifier (HRID)

The following sections describe the `hrid_root` component, i.e. the `hrid` minus the version identifier.

```
hrid_root:   archetype_hrid_root
             | template_hrid_root
             | subset_hrid_root
             | query_hrid_root
```

3.2.2 Archetype Identifier

The archetype human-readable identifier consists of two logical parts: an identifier of the reference model (i.e. logical information model) class on which it is based, and an ontological identifier.

The identifier is defined by the following grammar rules, which are a slightly simplified version of the grammar for the openEHR / ISO 13606 `ARCHETYPE_ID` type:

```
archetype_hrid_root:      qualified_rm_class_name '.' concept_id
qualified_rm_class_name: rm_publisher '-' rm_closure '-' rm_class
rm_publisher:             V_ALPHANUMERIC_NAME
rm_closure:               V_ALPHANUMERIC_NAME
rm_class:                 V_ALPHANUMERIC_NAME
concept_id:               V_SEGMENTED_ALPHANUMERIC_NAME

V_ALPHANUMERIC_NAME:      [a-zA-Z][a-zA-Z0-9_]+
V_SEGMENTED_ALPHANUMERIC_NAME: [a-zA-Z][a-zA-Z0-9_-]+ -- allows hyphens
```

The field meanings are as follows:

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

rm_closure: identifier of the reference model top-level package closure on which the archetype is based;

rm_class: name of class or equivalent entity in the reference model on which the artefact is based;

concept_id: an identifier from an ontology of information artefacts (see below);

The first part takes the form of a 3-part identifier, such as:

```
openEHR-EHR-EVALUATION
ISO-ISO13606-ENTRY
```

This historically has been used in *open*EHR and CEN/ISO 13606-2 to identify the reference model class on which an archetype is based. It includes the publisher of the reference model (e.g. “ISO”, “*open*EHR”), which top level ‘closure’ is being referred to, and finally which class.

The notion of ‘closure’ is a top level package from which the focal class can be reached. In general, a given class can be reached from more than one top level package, but an archetype of that class will only be suitable for one of those packages. For example, the openEHR class `CLUSTER` is used by classes in both the `ehr` and `demographic` top level packages. However, an archetype of `CLUSTER` will usually be designed for use with only one of those packages. The Cluster archetype `physical_examination` for example will only make sense in data defined by the `ehr` package. Consequently, it will have an archetype identifier of the form `openEHR-EHR-CLUSTER.physical_examination`.

The closure part of the identifier could be used by tools to ensure for example that an ‘EHR’ `CLUSTER` archetype was never attached to a ‘demographic’ information item.

3.2.2.1 Concept Identifier

The second part of the human-readable identifier is a ‘short’ ontological identifier (known in ADL 1.4 as the ‘concept’ or ‘domain concept’). Such identifiers have historically been natural language words or phrases, typically in a short mnemonic form, e.g. ‘bp_measurement’ in the archetype identifier `ISO-ISO13606-ENTRY.bp_measurement.v1`.

Legacy ADL 1.4 Semantics

Historically in ADL 1.4 (ISO 13606-2:2008), the ‘concept’ part of the identifier encoded the specialisation hierarchy of concepts as a series of hyphated segments, e.g. ‘problem’ and ‘problem-diagnosis’, with the latter identifiying a specialised form of the former.The requirement for the concept name to include specialisations is removed in this specification, as well as the ADL / AOM 1.5 specifications. This enables the domain concept of any artefact to be freely assigned according to the purpose of the artefact.

To allow for the fact that legacy specialised archetypes do in fact include the '-' style of separated domain concept identifier, the '-' character is still be allowed, but no longer has any semantic significance.

One consequence is that for archetypes with identifiers conforming to this specification, the level of specialisation can no longer be determined from the identifier. This new approach is in line with how source artefacts are named in object-oriented languages.

Concept Identifier Semantics

The more important aspect of the concept identifier, is its origin and semantics. Historically it has been part of the identifier for archetypes because it is human readable and facilitates debugging of systems where the data contain such identifiers. Clearly a purely *ad hoc* assignment of a human-readable identifier is not scalable or reliable. Consequently rules and mechanisms for assignment need to be identified.

This specification takes the point of view that the concept part of a *managed* knowledge artefact identifier must come from an ontology corresponding to the namespace of the identifier, in other words, an ontology maintained by a Managing Organisation or some higher authority.

It is not the business of this specification to define the ontology, but we can indicate the general form as being an ontology of *information entity types for use in the domain of health*. It is assumed that there are nodes within the ontology are related to the classes from the information (i.e. ‘reference’) model. This leads to an ontology of the form shown below.

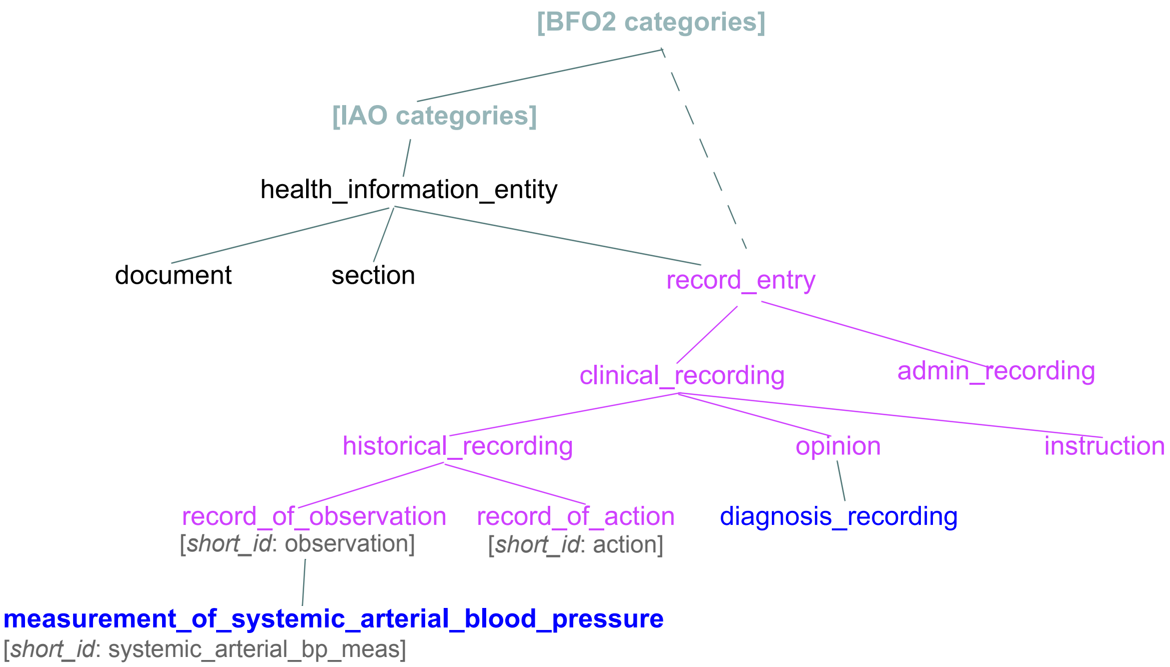


FIGURE 3 Example archetype concept ontology within a namespace

This (putative) ontology consists of high-level health information recording entities (black), a set of record entry types derived from the Clinical Investigator Record ontology (Beale and Heard, MedInfo 2007)[1], and domain-specific entities in blue. It is assumed that the top node(s) of the ontology could be related to nodes in a published ontology such as the Information Artefact Ontology (IAO)[2], but this is not a pre-requisite for establishing this ontology. More ideally, its categories would be related to categories in the Basic Formal Ontology (BFO 2)[3].

The blue node **measurement_of_systemic_arterial_blood_pressure** (bottom left) describes an entity corresponding to a 'record of systemic arterial blood pressure measurement'. Long names such as this are standard in the ontology community, and are designed to ensure that the name of a category is sufficient to unamiguously define its meaning. Such names are typically too long and unwieldy for the purposes of managable lexical identifiers such as for archetypes.

We therefore assume that a system of 'short identifiers' is possible within the ontology, where a 'short id' is a synonym for a full node identifier. If we further assume that the ontology is constructed with tools (e.g. Protege[4]) and that ontology identifiers are checked to ensure uniqueness.

Facilities to manage such ontologies should be available either centrally (e.g. openEHR.org, OBO[5]), so that every added archetype, template or subset is assigned a short ontological identifier from the ontology.

Existing archetypes can be accommodated within such ontologies in two possible ways. If they have been in use, and data exist containing these identifiers, then their current ontological identifiers can be proposed as the short id for an ontology class defined for the archetype. If there is a clash, a new

[1] http://www.openehr.org/files/publications/health_ict/MedInfo2007-BealeHeard.pdf
[2] https://code.google.com/p/information-artifact-ontology/
[3] https://code.google.com/p/bfo/
[4] http://protege.stanford.edu/
[5] http://www.obofoundry.org/

archetype concept short identifier will be needed, and the archetype will need to be republished under a different identifier.

3.2.2.2 Need for RM Class Name in Identifier

Theoretically, the Reference Model class identifier part (`qualified_rm_class_name` above) should not be needed in a well constructed identifier, on the basis that there should never be a clash of concept identifiers, regardless of the RM class, even though they can easily be similar. For example, a reasonable concept_id for an ENTRY (ISO 13606) or OBSERVATION (openEHR) structure archetyped to represent a generic lab result result might be ‘lab_result’. For the COMPOSITION-level archetype designed to contain any ‘lab result’ ENTRY / OBSERVATION, a reasonable name would typically be ‘lab_report’ (or the equivalent in another language).

Unfortunately, for some informational concepts, the appropriate name for the actual core data level can appear to be perfectly reasonable also as a name for a higher level container of the same data. Without an efficient and essentially global ontology construction service or authority available, the inclusion of the qualified RM class name acts as a reasonable guard against such clashes.

If in the future a capability becomes widely available for efficiently defining ontology concept identifiers for archetypes, the archetype identifier could be reduced to a purely namespaced and versioned ontology identifier. Such an identifier would resemble the following example:

```
org.cimi::chem7_panel_result.v2.0.4
```

3.2.3 Template Identifier

Within a given publishing space, template human-readable identifiers are defined the same way as archetype identifiers, i.e.:

```
template_hrid_root: qualified_rm_class_name ‘.’ domain_concept
```

3.2.4 Terminology Subset Identifier

Terminology subsets (aka ‘ref-sets’, i.e. ‘intentional reference sets’ as defined by IHTSDO) are a relatively new type of artefact. The key requirement is that a system of terminology subset identifiers accommodates multiple any terminology, regardless of its coding system, publisher or internal design.

A possible proposal for a subset identifier is to use the ontology approach above, within a larger identifier constructed as follows:

```
subset_hrid_root:          qualified_terminology_id '.' concept_id
qualified_terminology_id:  terminology_originator ‘-’ terminology_name
terminology_originator:    V_DOMAIN_NAME
terminology_name:          V_ALPHANUMERIC_NAME
```

TBD_1: these definitions probably need to be aligned to CTS2 expected definitions, assuming the latter can be used for any terminology, regardless of pubblishing agency.

This would lead to identifiers like the following:

```
org.ihtsdo-snomed_ct.blood_phenotype.v2 -- Snomed Blood type subset
int.who-icd10.bacterial_infections.v13 -- ICD10 bacterial infections subset
```

In the above, the concept_id is a short form of an ontological identifier for the ref-set or value set.

TBD_2: An alternative is the SNOMED CT system, whereby reference sets are identified using SctIds, i.e. SNOMED concept codes. It is unclear how this would be made to apply to terminologies other than SNOMED CT and the small number of terminologies that have been mapped to it however.

3.2.5 Query Set Identifier

There has been little experience with identification of query sets as a design artefact, mainly because queries in most systems are written in SQL and are not portable to any other system, being based on the local database structure.

Archetype-based queries, written in AQL or a similar formalism are portable across systems, and therefore do not need to be re-designed for each environment. Their identification is therefore likely to be of far greater importance than that of non-portable queries.

TBD_3: human-readable id for queries

3.3 Versioning

3.3.1 General Model

Unlike software artefacts in most modern versioning systems, knowledge artefacts are *individually version-controlled*. This is because an archetype, template or terminology subset is, in and of itself, a potentially complex structure of data points / groups and / or terminology codes and relationships. It can in general be used on its own or with a small number of related artefacts (e.g. specialisation parents). Therefore, the version identification system applies to *each source artefact*, rather than an entire repository in the manner of typical software versioning.

This has a very visible effect: it means that every ‘committed’ change to an artefact is like a release, whereas with software, numerous changes to source files typically occur between releases. Additionally, each artefact revision is *distinguished by its version identifier* for the purpose of change tracking in a repository environment, whereas with software source artefacts, the logical ‘name’ of each entity (e.g. a class called ‘LinkedList’) within the source repository doesn’t change, even though its contents do. To summarise:

- software versioning is performed by successive snapshots of a repository, and releasing is performed by assigning a version identifier to some of the snapshots;
- for knowledge artefacts being described here, versioning occurs *independently* for each artefact, and ‘releasing’ is simply an act of publishing the artefact;
- for knowledge artefacts, the versioned human-readable identifier is or can be used computationally, e.g. in queries and artefact references, whereas a software release identifier is not generally computed on by the software itself.

3.3.2 Version Numbering

Despite the above differences, the numbering of versions of knowledge artefacts follows nearly all of the accepted rules for identifying software releases, as described by semver.org.

Accordingly, version identifiers are based on three levels of ‘versioning’, identified by dot-separated numeric parts, with an optional extension related to the artefact lifecycle, described below. The numeric parts are:

- **major version** - *must* be incremented with a breaking change to the artefact *formal definition*; *may* be incremented with a lesser change;
- **minor version** - *must* be incremented with a non-breaking change to the artefact *formal definition*; *may* be incremented with a lesser change;
- **patch version** - *must* be incremented with a change to the informal parts of the artefact;

- **commit number** - a number that is incremented every time an artefact is committed; can be used on its own to uniquely identify any version of an artefact. This number is never reset, and thus at any point in time indicates the total number of committed changes over the lifetime of the artefact within its current publishing domain.

In the above, the 'formal definition' refers to the following parts of an archetype or template only:

- the identifier section;
- the `specialize` clause;
- the `definition` section;
- within the `terminology` section:
  - the `text` short names of the terms in the `term_definitions` section (i.e. not the description long text or other meta-data);
  - the `term_bindings` section;
  - the `value_set` section.

Lexically, the version identifier is defined as follows:

```
version_id:         release_version [extension]
release_version:    major_version '.' minor_version '.' patch_version
major_version:      {V_NUMBER}+
minor_version:      {V_NUMBER}+
patch_version:      {V_NUMBER}+
version_extension:  version_modifier commit_number
version_modifier:   '-rc' | '+u'
V_NUMBER:           [0-9]+
```

This leads to identifiers such as:

```
1.3.5
1.3.5-rc174         # release candidate for version 1.3.5, commit 174
1.3.5+u22           # unstable development version based on version 1.3.5
```

The following general rules are required for using version identifiers.

- *First version rule*: the first version (i.e. version on creation) of an artefact is a 'v0' version, i.e. 0.N.P. Usually it is 0.0.1, but may be a higher v0 version to indicate maturity. The discussion of lifecycle and distributed semantics below provide more details on the initial version semantics.
- *Incrementing rule*: when generating a release version (i.e. not a candidate or unstable version), when the major version is incremented, the minor and patch version numbers are reset to 0; when the minor version is incremented, the path number is reset to 0.

More specific rules relating to specific lifecycle states are described below.

Two 'variant' versions are defined in the above syntax: 'release candidate' and 'unstable'. The first is a standard software classification, syntactically indicated with the tag 'rc'. Version numbers including 'rc' are always of the form 'M.N.P-rcC', e.g. '1.3.5-rc189', where the minus sign ('-') is understood as indicating a version that is 'less than' the target version '1.3.5', i.e. '1.3.5-rc189' is an interim version leading to the stable version 1.3.5.

The other variant is indicated with the modifier '+uC, where '+' indicates a version 'after' the version identified by the preceding numeric identifier, 'u' indicates an 'unstable' development version, and C represents the commit number. An unstable version is assumed to include **changes of any magnitude** with respect to its base version. This is a departure from the semver.org rules, but is necessary for perartifact identification, since an artefact has an identifier at all times. The use of '+u' version identifiers is described further below.

Note that only the major version forms part of the source artefact *human-readable* identifier. The intention of that is that a breaking change causes a new artefact from the point of view of deployment. This is analagous to breaking changes in software interfaces, web service defintions etc, being seen as a distinct entity, typically deployed *alongside* the old version.

TBD_4: we might want to also allow post release build versions e.g. 1.3.3+28? So far this spec doesn't include this possibility.

3.3.3 Change Semantics

The semver.org model is designed for software, and is based on the concept of the software interface, or 'public API'. For the the artefact types within the scope of this specification, the concept of 'interface' is interpreted as being the .

A 'breaking change' for knowledge artefacts in the scope of this specification is defined as follows:

- for archetypes and templates, any change that prevents data created by the previous release of the artefact validating against the new release.
- for terminology subsets, any change that causes coded data to no longer be found in the relevant subset in the owning model (i.e. archetype or template).

Examples of breaking changes are:

- removal of mandatory data points or groups;
- move of data points to different sub-tree.

Any such change necessarily requires a new major version. The logical consequence of these rules is that non-breaking (minor version) changes can include:

- constraints redefined to be 'wider' (i.e. old constraint subsumed by new constraint);
- additional model nodes (i.e. extensions).

This has the important side-effect that minor versions of a given major version may have additional semantics comared to the original major version (i.e, minor version 0) and any other intervening minor version. In other words, **specifying a major version in general may not be sufficient to designate all of the 'interface' available in the latest minor version**. Therefore, for purposes of referencing an artefact with the expectation that the reference will designate specific elements, at least a minor version may be needed. This is discussed further in section 6.

Note that there is no assumption that a change of a given technical level (i.e. as evaluated by a diff tool) will be seen equivalently by domain experts. For example a minor change that only requires the patch version to be incremented might have major implications for clinical semantics. For this reason, the version identifier may be incremented beyond the minimum level required by a mechanical comparison.

4 Lifecycle Model

4.1 Conceptual Model

As with software, knowledge artefacts follow a lifecycle with identified states, representable by a state diagram. A traversal through the state diagram corresponds to the development of changes to an artefact leading to a release of a specific version.

Although somewhat peripheral to the scope of artefact identification, we describe a 'de facto' lifecycle for three reasons:

- to provide at least one lifecycle definition for users who have no other definition available;
- to provide explicit terminology for states and transitions for use in this and other specifications;
- to concretise the relationship between versioning and state transitions that commonly occur in software and other formal artefact development.

The lifecycle defined here is shown in FIGURE 4.

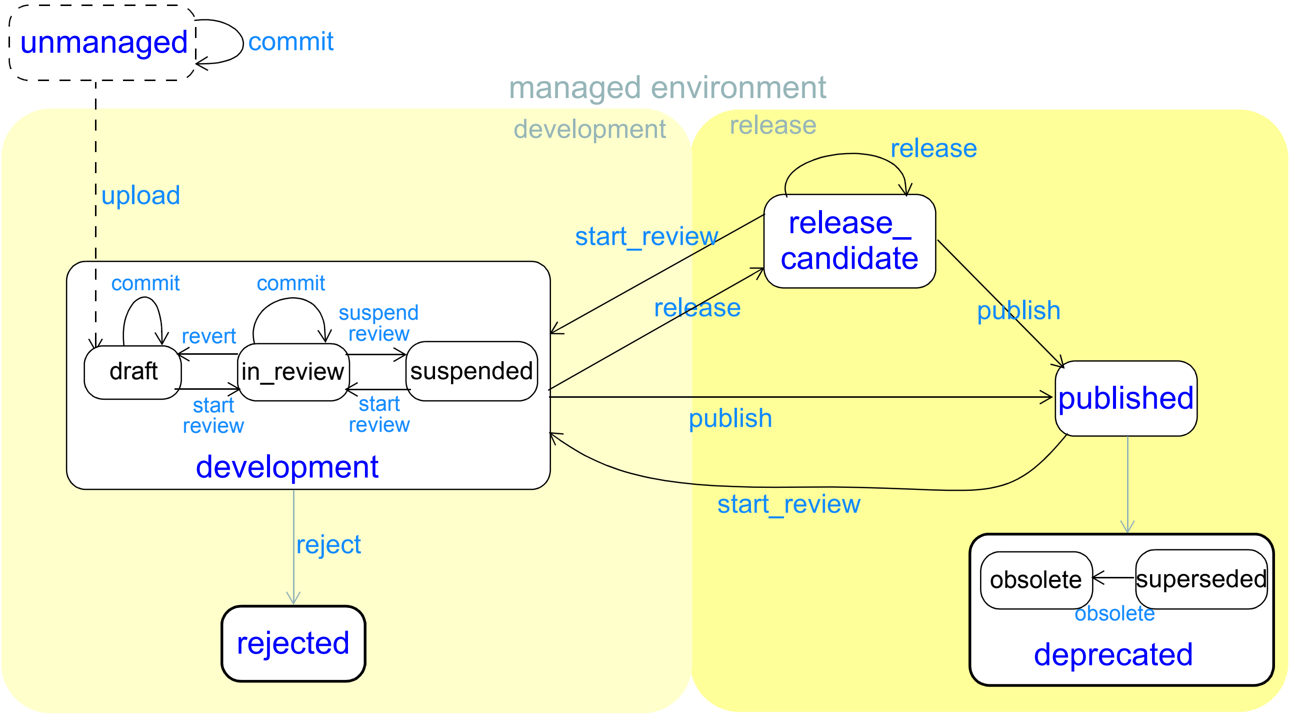


FIGURE 4 Development Lifecycle

A multi-level model is used, where some states have 'micro-states', and top-level states are known as 'macro-states'. The intention is to provide standard names for all macro-states, while suggesting and allowing micro-states where they make sense. Macro-state names are the basis for software version identification - 'development' corresponds to the '+u' variant, release_candidate to the '-rc' variant. Micro-states are useful to indicate because they define names for finer-grain states typically supported in artefact repositories.

This lifecycle assumes that artefacts start life either in an ‘unmanaged’ environment or directly in a managed one. In the latter case, it is assumed that there is some distinction between the developers’ view and the ‘release’ view.

The key states are defined with names (dark blue) and transitions (light blue) that correspond to typical software and document development terms. Typical traversals through the lifecycle are:

- (unmanaged =>) development => published
- development ... development => release_candidate => ... release_candidate => published
- published => deprecated
- development => rejected

A few linguistic conventions used here are worth noting:

- ‘start_review’ is the name of all actions entering the ‘development’ macro-state;
- ‘release’ as an action (i.e. state transition) is taken to mean making any version of an artefact available to the public user base, including pre-releases, final releases and post-releases (‘builds’ in semver.org parlance);
- ‘publish’ as an action means to make a definitive release.

4.2 Lifecycle-based Versioning

The correspondence of versioned human-readable identifier and lifecycle states can now be described, according to the illustration below.

The version identifier evolves according to the general rules described above, and specific rules related to the lifecycle states, as follows.

- An artefact normally starts life at ‘0.0.1’, although it is acceptable practice to start at some other v0 version e.g. ‘0.5.0’ to indicate approximately how mature the artefact is. It remains as a v0 version for a period of unstable early development leading to an initial releasable ‘1.x’ version.

  Technically, v0 versions should be appended with ‘+uC’ according to the above rules, but as this is unlikely to be univesally supported in tools, and the whole point of v0 is to indicate instability, it is recommended only for tools communicating with artefact management environments that support this versioning scheme.
- At some point, the artefact will be uploaded to a managed repository, at which point its identifier will be prepended with the management organisation namespace (and may change in other ways).
- During active development, an artefact is considered to be *unstable*, i.e. any kind of changes may be made, reversed, redone and so on; due to this, the version id is formed from the *last stable based version id*, and appended with ‘+uC’ where C is the current commit number; for each cycle in this state, only the commit number is incremented;
- An artefact may be rejected, in which case its minor version is incremented (following the semver.org rules), and the artefact lifecycle state is set to ‘rejected’.
- At some point, the authoring team of an artefact will decide the artefact is ready for release. Its release version id is then calculated as a function of the difference between the current form and the base version on which it is based.
- It is then either:

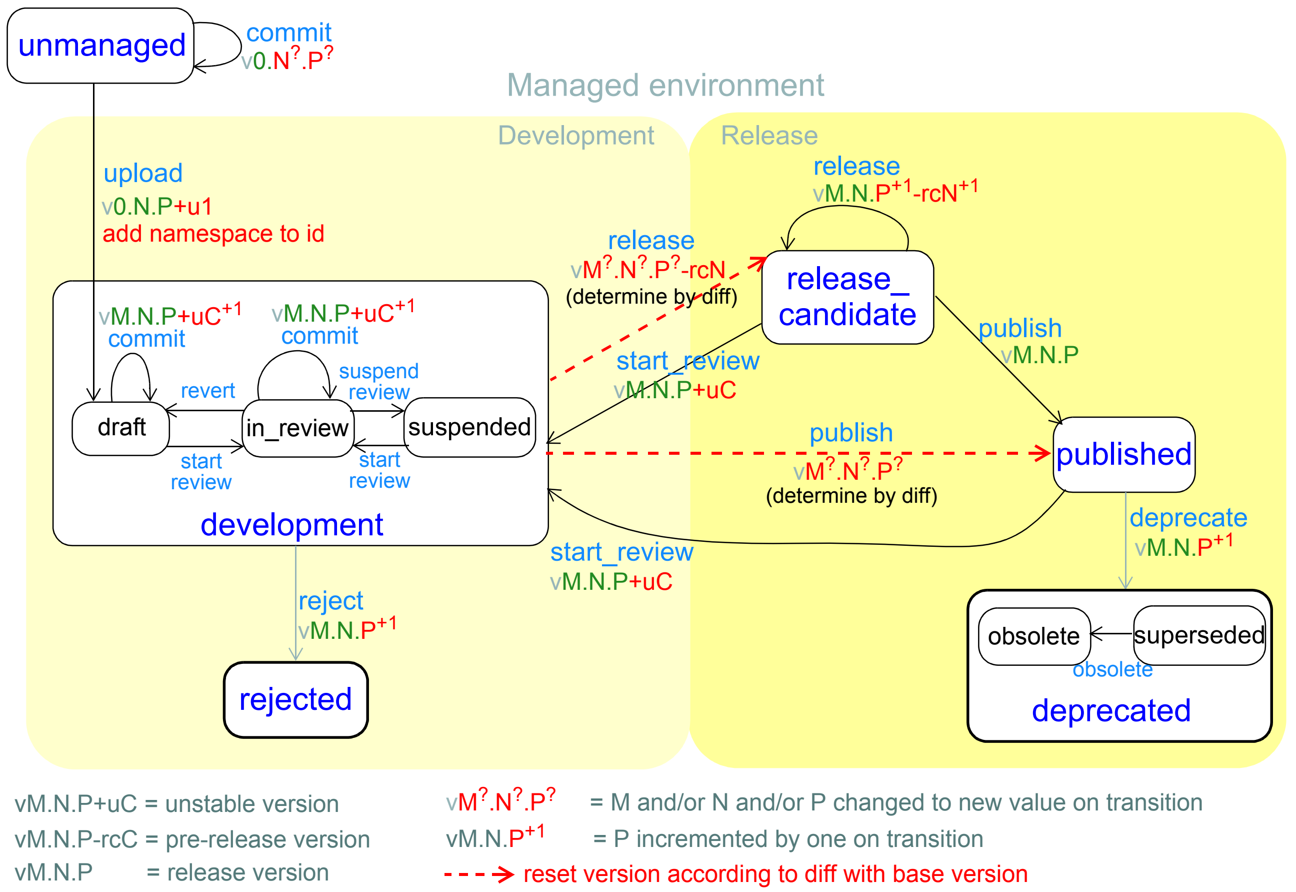


FIGURE 5 Development Lifecycle and Versioning

- published into a pre-release cycle, at which point the numerical part of the version is computed according to the difference between the current form of the artefact and the release version on which it is based. The form of the identifer becomes M.N.P-rcN, which indicates a community testing phase. From the `release_candidate` state, three paths are possible:
  * publish definitively with a stable version id of the form M.N.P;
  * release a newer release candidate, containing only changes that do not break the interface, i.e. patch-level changes or less; increment the commit number in any case;
  * if larger changes are needed, go back into the ‘development’ state and perform larger changes as needed;
- released directly to a stable version of the form M.N.P.

- An artefact may eventually be deprecated, in which case its minor version is incremented (following the semver.org rules), and the artefact lifecycle state is set to ‘deprecated’. It may be classified inside the repository registry as ‘obsolete’, ‘suspended’ or in some other way.

According to the basic version rules and the lifecycle model above, the ‘precedence’ of version identifiers follows is exemplified by the following:

```
    1.2.3-rc18 < 1.2.3-rc24 < 1.2.3 < 1.2.3+u67 < 1.2.3+u108 < ... < rel_version
```

where ‘rel_version’ could be 1.2.4, but may be something higher, depending on the differences between the final ‘+u’ version and version 1.2.3.

5 Distributed Governance

5.1 Overview

This section deals with how knowledge artefact identifiers are managed in the distributed environment illustrated in FIGURE 1. Rules are needed to define how identifiers are managed in the event of an artefact coming under management, as well as transfers and forking of managed artefacts.

5.2 Management

Many knowledge artefacts start life in an *ad hoc* way, created by a research project or expert individual. From the point of view of this specification, they are initially ‘umanaged’, meaning they have no custodial organisation.

The first step to making an artefact widely visible, and usable to the outside world is to bring it under management of an organisation that follows rules of governance and quality assurance on which the outside world can rely. This specification does not describe all these rules, just the rules for identification and meta-data of artefacts coming under management.

When an artefact is first created, its lifecycle state is ‘unmanaged’ and its version identifier is v0.N.P, i.e. a ‘pre- v1’ version, generally recognised (including by `semver.org`) as being an unstable form of the artefact that makes no promises with respect to the normal major/minor/patch versioning rules. The artefact may be given a Guid by tooling, although this will be ignored by a management organisation due to the fact that Guids assigned by *ad hoc* tools or direct human authoring are often copies of existing Guids (due to cut and paste) or are unreliable in some other way (improper Guid algorithm implementation).

When an artefact is accepted by a Managing Organisation, the following things happen:

- its lifecycle state progresses to ‘initial’;
- its human-readable identifier is changed to the namespaced form;
- it is assigned a newly generated Guid as its uid;
- if its major version number is higher than 0 it is reset to 0.0.1, otherwise it is left unchanged;
- various meta-data items are set, including copyright, license.

In addition, a SHA-1 hash may be generated for the artefact, which is stored within the repository.

5.3 Virtual Referencing across MOs

To Be Continued:

5.4 Transfer and Forking

Once an artefact is under management, it evolves according to the lifecycle described earlier in this specification. Most of these steps and transitions can be considered ‘details of development’. However, when an artefact is deployed, data will be created containing the artefact identifiers, and from this point, the ability to link data to the generating artefacts reliably is the critical issue. The standard approach to this is described in the next section.

Challenges in data / artefact identification arise from the transfer and/or ‘forking’ of artefacts among Managing Organisations. Artefacts can have two possible roles in a management organisation:

- as actively developed and maintained artefacts;
- as deployment artefacts.

A Managing Organisation may decide to cease its own maintenance of an artefact, and transfer that responsibility to another organisation, e.g. a national level MO. Usually it will continue to use the current local form of the artefact in its current deployment contexts, e.g. by local hospital systems or vendors.

At the moment of acquisition by the new MO, the artefact's HRID would potentially be re-assigned.

At some point the new custodian will perform maintenance work on the artefact, for example releasing a new minor or patch-level version. If such new releases are considered national standards, the *original* MO will most likely adopt them for use. The question is: how are the new releases of the artefact identified?

With respect to the human-readable identifier, two basic strategies are available: retain the original human-readable identifier, or change it to reflect the new MO. An argument against changing it is that identifier continuity would be preserved, ensuring that archetype references in extant queries and in data, as well as in other archetypes and templates remain valid. If it is assumed that all such references are limited to the original management domain, the size of this problem is known and most likely containable.

Arguments for changing the identifier include:

- a requirement of the new Managing Organisation to be identified in the artefact; this may be a global expectation of industry as well, e.g. if the new manager is a national organisation, it will clearly be easier for vendors and system managers if the artefacts it releases carry its identifier;
- the possibility that the original domain continues to create new local releases, perhaps in response to problems experienced locally that require unavoidable locally specific changes;
- the new MO wants to rename the artefact to fit in better with its own ontological artefact classification;
- if no data or queries have ever been created using the artefact in question, changing its identifier will have no concrete impact anyway;
- if the namespace always reflects the current MO, it will be easier to know who to contact for support and other purposes.

The second of these points constitutes a 'fork' in software terms, i.e. one line of development becomes two. Common sense would seem to dictate that the likelihood of forking, particularly due to the unforeseen need of dealing with local problems after an artefact has been promoted to a higher management domain, will never be zero, and that it may even be frequent.

It also seems reasonable to assume that even if there were no rule or obligation to change the identifier of an artefact when it migrates from one manager to another, that it will occur by mutual consent in some situations anyway.

The approach of this specification is therefore that rules must be provided that define how artefact re-identification can be effected, without actually requiring it to be done in any particular situation. Part of the requirement is to establish a machine processable concept of 'artefact equivalence'.

Rules for migration are required for both the human-readable identifier and the machine identifier. With respect to the human-readable identifier, any of the following are assumed to be mutable:

- *namespace*: at a minimum this will always change;

- *concept_id*: the ontological identifier may or may not change, depending on whether the new manager wishes to locate the artefact in a different ontology;
- *version identifier*: the version identifier will in general change, possibly as a function of whether the concept part of the identifier changes.

The general case is that the transfer of an artefact to another management organisation *may* result in an identifier that changes in all aspects apart from the reference model related parts of the identifier, which cannot change for formal reasons.

It is assumed here that when the human-readable identifier changes (no matter how minimally), the uid property must be changed as well. This is to prevent confusion between subsequent new versions of the original with releases of the transferred artefact. A new uid is further justified by the unavoidable ‘migration is forking’ assumption.

To enable tools to determine what archetypes are equivalent, a specific section of the artefact meta-data is proposed, which records the equivalence between the current identifier and previous ones. Assuming that an artefact could migrate more than once in its life, this section would need to accommodate multiple such statements. For purposes of helping human use of this information, it is also proposed that a date be included. The section would therefore have the logical structure of a history of equivalences, as shown in the following example for an archetype (‘hrid’ = human-readable id):

```
id_history = <
    [“2001-05-27”] = <
       old = <
          hrid = <“au.com.rbh::openEHR-EHR-EVALUATION.problem_desc.v2.4.1”>
          uid = <“5221C9E5-0ECA-469F-83C5-A5D5A0C6682C”>
       >
       new = <
          hrid = <“au.gov.nehta::openEHR-EHR-EVALUATION.problem.v1.0.1”>
          uid = <“094C8B37-F0CD-45C9-A1B7-CDFDE14C67AB”>
       >
    >
    [“2004-14-03”] = <
       old = <
          hrid = <“au.gov.nehta::openEHR-EHR-EVALUATION.problem.v1.6.3”>
          uid = <“E50290BB-890A-4344-9480-D40AF01C5BCC”)
       >
       new = <
          hrid = <“au.gov.doha::openEHR-EHR-EVALUATION.problem.v1.6.3”>
          uid = <“F4166F58-4EDA-4F13-B413-45A8F7A3E53D”)
       >
    >
>
```

These equivalence histories would be used by Managing Organisations to populate artefact identifier equivalence tables that could be shared on request with other manager organisations. This system is reminiscent of the CNAME record type in the internet Domain Name System (DNS), which is used to record alias domain names for canonical domain names.

6 Referencing

This section describes how artefact are referenced, by other artefacts and software. The general principal for referencing is that references based on human-readable identifiers are used between source artefacts, in the same way as for software, while references in operational forms of the artefacts or from data may be in the form of either HRIDs or machine identifiers.

A key semantic difference exists with references as opposed to identifiers. A reference is either a full physical artefact identifier, or else an identifier with partial version information. In both cases, a reference is used to *match* artefacts carrying full identification. In general, there can be several candidate matches, and therefore a matching algorithm has to be specified in each case, in order to ensure constant meaning for a given reference in all modelling and computing environments.

Various forms of the references based on the HRID are used, depending on the need, as described below. These are denoted as follows:

- the *interface* HRID reference (ihrid_ref), which is the same for all artefact instances sharing the same core interface, in other words, the form of the identifier including only the major version; this will match the latest relase available of that major version;
- the *specific interface* HRID reference (sihrid_ref), which is the form of the identifier including the major and minor versions, which will match a specific release of the interface;
- the *physical* HRID reference (phrid_ref), which identifies artefact instances which are identical; this is the form of identifier with the full version identifier included.

The grammar for this is as follows.

```
hrid_ref:            namespaced_hrid_ref | local_hrid_ref
namespaced_hrid_ref: namespace '::' local_hrid_ref
local_hrid_ref:      ihrid_ref | sihrid_ref | phrid_ref
ihrid_ref:           hrid_root '.v' major_version
sihrid_ref:          hrid_root '.v' major_version '.' minor_version
phrid_ref:           local_hrid
```

By way of example, the following two archetype iHRID references denote different logical archetypes, from a data processing point of view, since their major versions are different, indicating a breaking change between the two:

```
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v2
```

Conversely, the following references denote physically distinct archetypes that are regarded as logically substitutable:

```
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.1.5
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.1.7
```

6.1 Source Artefact References

This section describes the scenarios and representation required for identifier-based referencing between design time 'source' (i.e. compiler input) artefacts.

6.1.1 Archetype External References (ADL/AOM 1.5)

In ADL 1.5, a direct archetype-archetype reference, known as an 'external reference' can be defined, which uses the archetype miniHRID, as shown in the following example.

```
ACTIVITY[id2] ∈ {-- Medication activity
```

```
            description ∈ {
                use_archetype ITEM_TREE [openEHR-EHR-ITEM_TREE.medication.v1]
            }
        }
```

If such a reference does not include a namespace, the meaning is that the same namespace as the current archetype is assumed (which may be no namespace). A namespace would be included to express a reference to an archetype outside the current namespace.

A reference such as the above will resolve to an actual archetype at runtime according to the following algorithm:

- the most recent *released* version variant of openEHR-EHR-ITEM_TREE.medication.v1, i.e. the latest minor or patch version such as v1.0.4, v1.2.49 etc OR
- the latest *release candidate* version of openEHR-EHR-ITEM_TREE.medication.v1, e.g. v1.2.3-rc44, because 'rc' versions are guaranteed to be semantically compatible with their target version

Because minor versions can include structural additions, it may be that in some cases, archetype external references need to be include a minor version as well - allowing an exact structural form of the archetype (interface) to be identified. Similarly For testing or other research purposes, it should probably be assumed that patch and '-rc' and '+u' versions of an archetype need to be referenced.

The general case is therefore assumed to be that an artefact reference is a hrid_ref, but most commonly an ihrid_ref.

6.1.2 Template References to Archetypes and Templates

Templates are normally designed as pre-cursors to software artefacts, such as forms, message definitions and document schemas. Consequently, their exact contents and structure are usually carefully controlled by their developers, in the interests of stability. To achieve this, references from templates to other templates or archetypes need to be able to refer to any level of version of the target artefact. During a development phase, it may be that the template references are limited to major versions of the human-readable identifier, i.e. the ihrid_ref. At some point it may be the case that the minor version must be included as well. As noted above, this is because minor versions of archetypes can include structural additions, and therefore affect the structure of the final document / data-set etc. It may even be the case that patch level versions need to be identified, so as to ensure no changes whatever can occur in the template, even if upgraded versions of the source artefacts become available.

Such tight control is not however a universal requirement. A conscious design decision may have been taken that says that the resulting software artefact contents are whatever results from the template definition at the time of publishing, assuming references to major versions only.

To accommodate these scenarios, template references to archetypes and other templates need to be legal at any version level. For example, any of the following references should be legal in a template:

- org.openehr::openEHR-EHR-EVALUATION.problem.v2
- org.openehr::openEHR-EHR-EVALUATION.problem.v2.4
- org.openehr::openEHR-EHR-EVALUATION.problem.v2.4.17

In development and research environments, it is reasonable to allow '-rc' and '+u' variants as well.

The general case is therefore as for archetype external references: a template reference is a hrid_ref.

6.1.3 Between Specialised Archetypes

A specialised archetype refers to its parent using the human-readable reference, *including only the major version*. Two possible variants can occur:

- With a non-namespaced reference. This is assumed to come from the same namespace as the specialised archetype.
- With a namespaced identifier where the namespace is different from that of the referencing archetype. This resolves against the latest release of the referenced archetype in the locally available repository copy of the referenced namespace.

The following figure shows a number of archetypes related by specialisation.

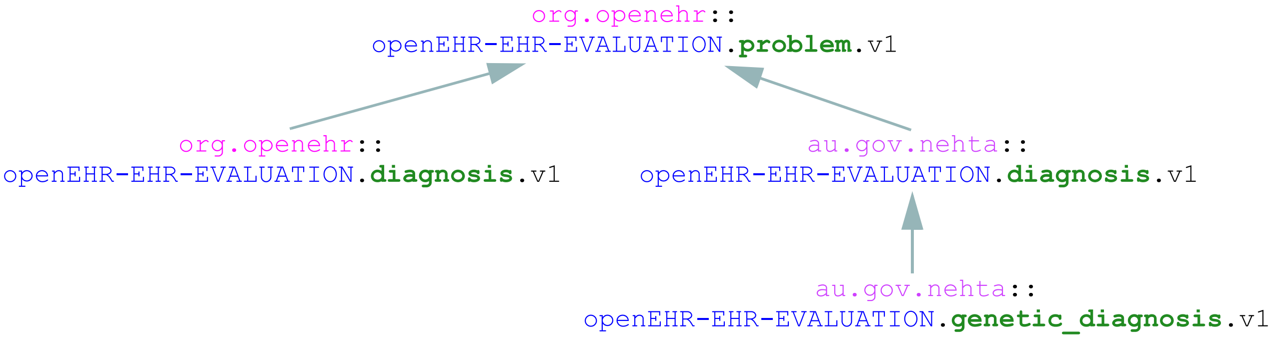

org.openehr::
openEHR-EHR-EVALUATION.problem.v1
org.openehr::
openEHR-EHR-EVALUATION.diagnosis.v1
au.gov.nehta::
openEHR-EHR-EVALUATION.diagnosis.v1
au.gov.nehta::
openEHR-EHR-EVALUATION.genetic_diagnosis.v1

FIGURE 6 Specialisation relationships

One question that naturally arises to do with specialisation is what happens when the parent archetype is revised. The approach is the same as for object-oriented software: all archetypes in a given 'check-out' or release must always compile at any point in time to be valid. If a revised parent is introduced that invalidates any of its inheritance children, revisions must be made to the children before the repository becomes valid as a whole again. This means that a new version of an archetype in general may require child archetypes to be re-versioned as well.

6.2 Source Artefact Relationship Constraints

Related to the concept of 'references' is constraints that when evaluated at runtime, resolve to artefact identifiers. Two types are described here, which are the two kinds of archetype 'slot' definition.

6.2.1 ADL 1.4 Archetype Slots

In ADL 1.4, archetypes slots are defined via assertions in their slot statements. Although the specification allows for all kinds of possibilities, the only one in use is regular expressions (REs) on the archetype identifiers allowed to fill the slot. Current ADL 1.4 tooling supports REs on full (non-name-spaced) ADL 1.4 archetype identifiers, which include only the major version number, e.g.:

```
openEHR-EHR-EVALUATION.problem.v1
```

Note that such REs often include disjoint patterns, by using the form "id_pattern1|id_pattern2|id_pattern3".

A typical slot definition using REs based on such identifiers is as follows:

```
protocol matches {
    ITEM_TREE[at0015] ∈ {
       items cardinality ∈ {0..*; ordered} ∈ {
            allow_archetype CLUSTER[id20] occurrences ∈ {0..1} matches {
                include
                    archetype_id/value ∈ {/openEHR-EHR-CLUSTER\.device(-[a-zA-
Z0-9_]+)*\.v1/}
                }
            }
        }
    }
```

This slot allows any archetype named openEHR-EHR-CLUSTER.device.v1 or openEHR-EHR-CLUSTER.device-xxx.v1, which used the ADL 1.4 method of signifying specialised archetypes.

The rule for namespace inclusion is as for external references:

- no namespace means the same namespace as the current archetype;
- an explicit namespace means archetypes from that namespace.

As for external references, there is technically nothing to stop a slot RE being defined to refer to specific minor versions or builds of an archetype. The same rule applies: released archetypes should only include major versions.

6.2.2 ADL 1.5 Archetype Slots

In ADL/AOM 1.5 a semantic slot type will be introduced in which matching archetypes are defined in the form of a constraint on the archetype concept (and optionally namespace), reminiscent of the SNOMED CT post-coordination constraint syntax. This is shown in the following example.

```
allow_archetype CLUSTER [id4.1] occurrences ∈ {0..1} ∈ {
    include ∈ {True}
        archetype_id ∈ {
            ARCHETYPE_ID ∈ {
                namespace ∈ {...}
                concept ∈ {<< investigation_methodology OR
                               << investigation_protocol}
                ...
            }
        }
    }
```

The above kind of referencing relies on an ontological underpinning for the concept_id part of the human-readable identifier.

6.3 AQL Query Sets

AQL queries are in general authored in a ‘set’ in order to achieve a design objective, e.g. populate a report, screen, or for some analytical objective. Many are purely local in nature and may be considered ‘throwaway’. Others are carefully designed for needs like populating a clinical guideline or performing a standard computation. Within an archetyped framework, such query sets need to be indentified and managed in a similar way to other artefacts.

6.4 AQL Queries

Archetype-based queries contain archetype references and paths, and can also contain template identifiers and paths. Typical examples are the paths (in green) in the following query:

```
SELECT pulse
FROM EHR[ehr_id/value=$ehruid]
```

```
     CONTAINS COMPOSITION c
     CONTAINS OBSERVATION pulse[openEHR-EHR-OBSERVATION.pulse.v1]

    WHERE c/name/value='Encounter‛ AND
        c/context/start_time/value <= $endperiod AND
        c/context/start_time/value >= $startPeriod AND
        pulse/data/events[id6]/data/items[id4]/value/value < 60
```

The semantics of referencing in queries differ from those of the archetype-to-archetype form, due to the fact that references are normally followed by paths that refer to specific data points within the structure. For an AQL query to be correct, the path must exist in the archetype at the release matched by the reference. Since minor versions can add to the archetype ‘interface’ (i.e. add data points, and therfore paths, to the structure), a given path needs to reference the oldest archetype for which the path is valid. Consider the following path:

```
[openEHR-EHR-OBSERVATION.pulse.v1]/data/events[at0006]/data/items[at0004]/
value/value
```

For this to be valid, the path `/data/events[at0006]/data/items[at0004]/value/value` must exist within the *earliest* v1.x release of the archetype `openEHR-EHR-OBSERVATION.pulse.v1`, i.e. v1.0.0. If this path happened to have been added in a more recent minor release, the archetype reference would need to include the first minor version containing that path.

Once an AQL query processor can work with a valid path, it will match the following data:

- any instance of the data point at that path in the referenced archetype;
- any instance of a data point in a *congruent* path in a specialisation child archetype.

An example of a congruent path in a child archetype is:

```
/data/events[id6.0.4]/data/items[id4.1]/value/value
```

6.5 Operational Artefacts

Operational artefacts such as flattened archetypes and operational templates generated by compiler tools are built from source artefacts, including by reference resolution from within some source artefacts to others within the current repository of the local and imported artefacts. The particular versions of reference targets are determined by the contents of the configuration, and are thus a function of version management activities, in the same way as for software development.

When an operational artefact is generated from controlled source artefacts (i.e. within a Managing Organisation), it is possible to include the fine-grained revision information from the relevant source artefacts, so that the operational form describes exactly which set of source artefacts were used to produce it. The source artefact semantic signatures can also be included. This information can be included in a configuration section of the artefact. This would be expressed in ODIN (previously dADL) or an XML equivalent, and would list the 'configuration' of concrete artefact revisions used to generate the operational version.

The structure of a Configuration is as follows:

```
configuration: archetype_config template_config subset_config rm_release
archetype_config: { config_item }+
template_config: { config_item }*
subset_config: { config_item }*
rm_release: rm_name release_id

config_item: identifier [ revision_id [ commit_id ] ] [ signature ]
```

```
signature: CHARACTER_SEQUENCE
revision_id: V_INTEGER
commit_id: V_INTEGER
release_id: V_STRING
```

An example of the configuration of an operational template in a controlled environment (dADL format) is as follows:

```
configuration = <
    archetypes = <
       [1] = <
          id = <"org.openehr::openEHR-EHR-OBSERVATION.heartrate.v1.3.28">
          signature = <"23895yw85y0y0">
       >
       [2] = <
          id = <"au.gov.nehta::openEHR-EHR-EVALUATION.genetic-
                                                     diagnosis.v1.2.0">
          signature = <"98typrhweruhfd">
       >
       [3] = <
          id = <"org.openehr::openEHR-EHR-EVALUATION.problem.v2.4.0">
          signature = <"2rfhweiudfwieurfh">
       >
    >
    templates = <
       [1] = <
          id = <"au.gov.nehta::openEHR-EHR-COMPOSITION.vital_signs.v5.36.1">
       >
    >
    subsets = <
       [1] = <
          id = <"org.ihtsdo.general::cardiac_diagnoses.v18.1.0">
       >
    >
    rm = <
       name = <"org.openehr.rm">
       release = <"1.1">
    >
>
```

6.6 References from Data

6.6.1 Requirements

In knowledge-enabled information environments such as those built on the archetype principles, knowledge artefacts are used to control the creation and validation of data, with the effect that data eventually stored in such systems ‘conform’ to the relevant artefacts. In order to be able to further process (e.g. display, modify and query) such data, references of some kind to the knowledge artefacts must be stored in the data. The requirements for such references depend on where the data are found, broadly within two possible situations, namely data within operational systems (e.g. EHR systems) and data within ‘messages’, ‘extracts’, or ‘documents’ sent between systems.

Three requirements can be identified with respect to data within systems.

Reconstitutability: firstly, it must be possible to re-connect data with the archetypes, templates and subsets, used to create them. This implies that the major and minor versions at least are recorded in data, since a minor version may have an effect on structure.

Querying: secondly, it must be possible to know what archetypes (including major version), and therefore what path-sets can be used for querying data - given that this may well include parents of specialised archteypes, not just the archetypes used to directly create the data.

Optimisation: we can also assume that in a typical production system handling millions of health records, that the size of artefect identifiers embedded in data (especially if repeated) may be an issue, and that some kind of space optimisation may be required.

Within extracts or messages, the same requirements broadly hold, but could be better restated as follows.

Reconstitutability: it must be possible for the receiving system to be able to determine the relationship of each data element with the artefacts(s) used to create it, so that it can be correctly reconstituted in the receiver system environment.

Querying: for ensuring the correct functioning of querying, the extract or message should potentially carry sufficient archetype lineage information the archetypes used in the data to allow querying at the receiver, particularly if the latter wants to be able to query using more general parents (e.g. a 'problem' archetype rather than some specific diagnosis specialisation).

Optimisation: a reasonable trade-off between space optimisation and clarity of representation must be used, given that messages, extracts etc flow between heterogeneous systems.

6.6.2 Reconstitutability

The reconstitutability requirement means recording archetype and template identifiers on the relevant nodes in the data. A basic form of this has always been used in *open*EHR, such that at archetype root nodes, the archetype identifier and if relevant the template identifier is recorded, and at interior nodes, the at-codes are recorded (formally, the archetype identifier and at-codes are recorded in the LOCATABLE.*archetype_node_id* attribute of each data node). For example, in data created based on *open*EHR Releases 1.0.2 or earlier, the archetype identifier references are of the form:

```
openEHR-EHR-EVALUATION.diagnosis.v1
```

With the more sophisticated identification system described here, these archetype references need to include namespace, and full version identifier, i.e.:

```
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.29.0
```

References with no namespace will remain legal, since there should be no computational impediment to using uncontrolled archetypes and templates, e.g. in an experimental situation. The lack of minor and patch level version numbers should also be legal for non-namespaced identifiers, and be interpreted as meaning '0' in both cases, i.e. '.v1' means '.v1.0.0'.

6.6.3 Supporting Archetype-based Querying

Querying of data in *open*EHR systems is assumed to be based on archetype 'path-sets', i.e. the set of paths extracted from an operational (flat-form) archetype. The paths are a slight simplification of standard X-paths. Two querying methods have been described to date, AQL and a-path, both making this assumption (see openEHR wiki).

Based on this assumption, given an archetype X used to create data, the following archetypes could be used for querying:

- X, i.e. exact same version, revision & commit;
- any previous minor or patch variant of X;
- any of the specialisation parents of X;
- any previous minor or patch variant of any of the specialisation parents of X.

For non-specialised archetypes, the allowable querying archetypes can be deduced from the archetype reference recorded in the data. For specialised archetypes, the specialisation lineage can only be obtained from the operational form of the archetype, found in the template used to create the data. This would create a potential problem where for data imported from another site without the relevant template(s), the archetype lineage information was not available. This would prevent the query engine at the receiver system knowing how to query the data using even the more general archetypes in the lineage, that it may have access to.

To address this situation, one of the following strategies is required:

- include the configuration meta-data from the operational template(s) with the data when it is exchanged, i.e. in an EHR Extract.
- include archetype lineage information in the data itself. This could be a modified form of the identifier reference in the case of specialised archetypes to allow lineage information to be stored.

The second approach can be considered a generalisation of recording just the current archetype identifier, i.e. the ‘lineage’ for non-specialised archetypes evaluates to just that archetype id, and for specialised archteypes, it will be a list. This specification assumes that the second is used.

The simplest form of this would be as a list of operational identifiers, e.g.

```
au.gov.nehta::openEHR-EHR-EVALUATION.genetic_diagnosis.v1.12.9,
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.29.0,
org.openehr::openEHR-EHR-EVALUATION.problem.v2.4.18
```

6.6.4 Formal Model

A formal definition of reference catering to the above requirements is as follows:

```
archetype_data_ref: archetype_ver_ref { ‘,’ archteype_ver_ref }*
archteype_ver_ref: hrid_root ‘.’ version_id_ref
version_id_ref: ‘v’ version_id
```

6.6.5 Optimisations

In normal archetype-based data, both basic references and additional lineage information might be repeated throughout a given component, such as an *open*EHR or ISO 13606 COMPOSITION. Consider a COMPOSITION documenting problems & diagnoses of the patient, where each problem is recorded using the archetype

```
uk.nhs.royalfree.clinical::openEHR-EHR-EVALUATION.diagnosis.v2.15.0
```

whose lineage is:

```
org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.29.0
org.openehr::openEHR-EHR-EVALUATION.problem.v2.4.0
```

In this example, the archetype reference lengths are 66, 57 and 54 characters respectively, i.e. a total of 177 characters. Repeated say 5 times would give 885 characters of identifier meta-data for the COMPOSITION, whose main clinical data could easily be similar. Even in an XML-based storage system, various kinds of compression are used, the identifier reference overhead might be considered as an unacceptable fraction of the overall data storage requirement.

It is therefore worth considering various simple optimisations, while retaining clarity and comprehensibility in the data. The following ideas are currently intended to be limited to serialised forms of data. They would therefore only require changes to *open*EHR XML-schemas rather than the abstract reference model.

6.6.5.1 Identifier Aliasing

The most obvious optimisation is to use a set of variable references local to the data context, in this case an *open*EHR or ISO 13606 Extract. For example, at the top of the Extract, the following definitions could be made:

```
id01=uk.nhs.royalfree::openEHR-EHR-EVALUATION.diagnosis.v2.15.0,
    org.openehr::openEHR-EHR-EVALUATION.diagnosis.v1.29.0,
    org.openehr::openEHR-EHR-EVALUATION.problem.v2.4.0
id02=au.gov.nehta::openEHR-EHR-OBSERVATION.hba1c_result.v1.4,
    org.openehr::openEHR-EHR-OBSERVATION.lab_result.v1.18
etc
```

The identifiers 'id01', 'id02' etc would then be used in the data, reducing the identifier overhead by perhaps 50% in some cases. This possibility would be enabled by adding an attribute to contain the variable definitions at the top of the `EHR_EXTRACT` type in the *open*EHR Reference Model, and in equivalent classes in other models.

The use of such variables will slightly complicate querying and other data processing, since a query that returns part of a Composition would return data containing meaningless local variable names rather than proper archetype meta-data.

A second question to consider is whether any parts of the identifiers could be removed. For example, it might initially appear that the reference model and class identification could be removed altogether, since the data when initially created would seem by definition to be based on the reference model and class of the archetype. However, neither are guaranteed. Consider the following two cases which use archetypes based on a different reference model to create data:

- a data extractor that transforms source data, say in *open*EHR form, to a standard form, say in ISO 13606 form. The archetype identifiers embedded in the latter data will be the original *open*EHR archetype identifiers (the extractor does not create new archetypes to do its transformation work);
- a product that is directly based on another standard, such as ISO 13606 but uses the published library of *open*EHR archetypes.

Similarly, in the case of the class, the data may easily be based on a descendant (e.g. the `POINT_EVENT` class in *open*EHR) of the class mentioned in the archetype (e.g. `EVENT`).

We therefore assume that although some of the above assumptions might be available in very particular environments, they cannot be safely made in general, particularly since it can never be predicted where data may be shared.

6.6.5.2 Reference Compression

Nevertheless, it would be possible to go further in terms of removing repetition in the once-only declarations. For instance, a compressed form of the archetype lineage information could be constructed, whereby repeated sections in each subsequent identifier are replaced by a special character. The example above would become:

```
id01=uk.nhs.royalfree::openEHR-EHR-EVALUATION.diagnosis.v2.15.0,
    org.openehr::~.diagnosis.v1.29.0,
    ~::~.problem.v2.4.0
id02=au.gov.nehta::openEHR-EHR-OBSERVATION.hba1c_result.v1.4.0,
        org.openehr.ehr::~.lab_result.v1.18.0
```

The above syntax uses the '~' character in each identifier in the list to mean 'the missing parts are taken from the corresponding element(s) of the previous identifier in the list' (the inspiration is the use of the '~' in dictionaries to stand for the keyword). In this syntax, the concrete archetype used to create the data is guaranteed to appear first and in its entirety in the list.

Clearly in a particular system in which archetypes were only ever used from the same reference model as the system itself is built on, an even further reduced form of these references could be created. However, if the data were ever to be shared, such references would be in danger of being non-interoperable.

Whether the additional saving in space justifies the added complexity in parsing is debatable.

7 A Reliable URI for Knowledge Resources

There should be a standardised and reliable Uniform Resource Identifier (URI) for all released knowledge resources, in both source and operational forms. This may justify its own scheme-space, but is at least achievable within the normal http scheme-space.

To Be Continued:

8 Scenarios

This section describes typical scenarios to do with artefact development, deployment and querying.

8.1 Minor Version Upgrade

To Be Continued:

8.2 Major Version Upgrade

To Be Continued:

8.3 Templates using Archteypes and Subsets

To Be Continued:

8.4 Artefact Transfer / Fork

To Be Continued:

9 Artefact Authentication

In theory, revision information should be reliable, and no two physical knowledge artefacts should exist that are either identical but have different identifiers and/or revision information, or are different but are identified as being the same. However, in practical systems, such situations can occur due to uncontrolled artefact creation, uncontrolled copying, and errors in version management.

TBD_5: hashing on source v operational artefacts? Consider templates that don’t change but referenced archetypes that do.

9.1 Integrity Check

It is therefore useful to be able to determine whether two artefacts (usually purported copies or subsequent revisions) are the same or not, even if the version information is the same. This can be achieved by the use of a digital hash function (e.g. SHA-1, MD5), which generates a 'fingerprint' of the artefact. Two archetypes with the same hash value must be the same – hash functions generate a different result if even a single bit is different in the input stream. However, applying such functions to the typical file representation of an archetype or template will not usually have the desired result. This is because differences in white-space and non-significant ordering, which make no difference to the semantics – will still generate different hash values. Other semantically insignificant differences include changes to meta-data values, such as descriptions, etc may have been changed (e.g. to correct spelling, improving wording), and changes or additions to translations.

As a consequence, the input to a hashing function for the purpose of generating a semantic signature of an *open*EHR knowledge artefact must be some canonical form of the original literal artefact, that is impervious to differences of the above types while retaining differences that will affect computation with such artefacts. The integrity check process is illustrated below.

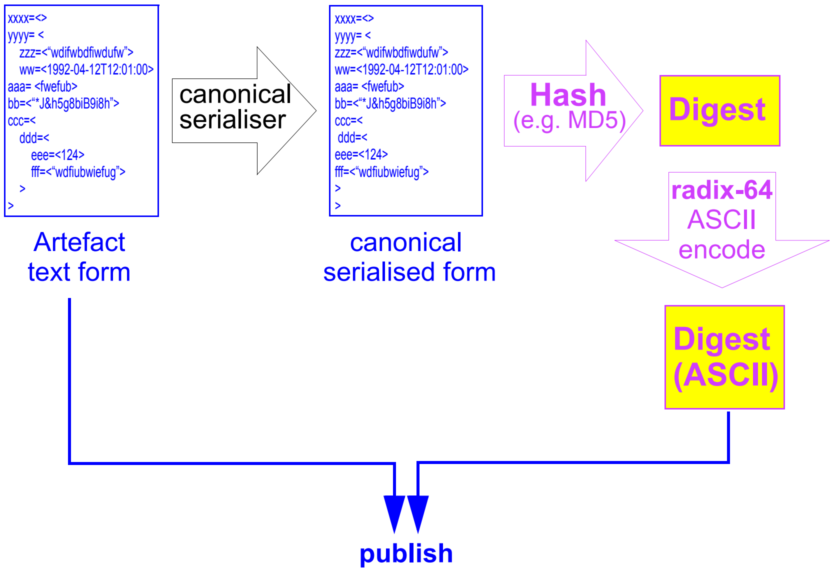
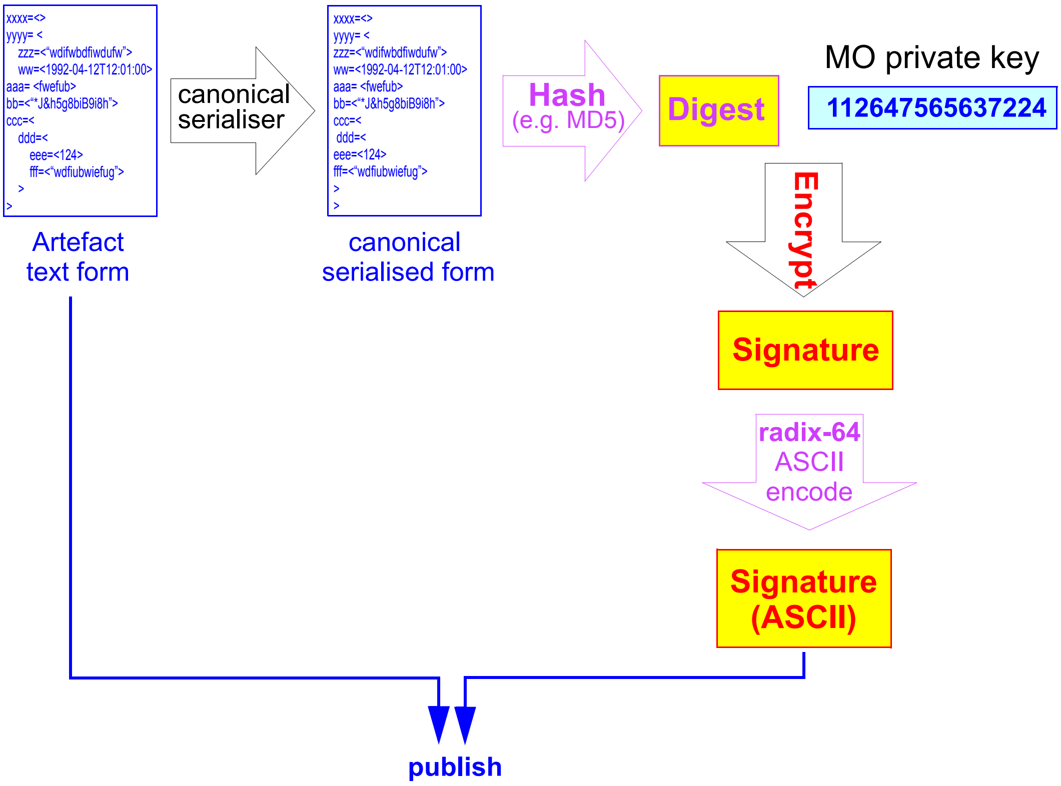


FIGURE 7 Integrity check applied to knowledge artefact

9.2 Authentication

A second need to do with validity of knowledge artefacts is establishing their authenticity, i.e. their true origin. The usual way of supporting authentication is with a digital signature. A typical scheme based on the public key infrastructure (PKI) concept is for the producer of an artefact to sign it with their private key, and for the public key to be used by a consumer of the artefact to decrypt the signed entity.

In the case of *open*EHR knowledge artefacts, the need is to know the originating Managing Organisation of an artefact. The PKI approach is for each MO to generate a key pair, and to provide the public key to the Central Governance Authority. Signing is then carried out using the MO private key on the hash digest already generated for an artefact. The modified process is illustrated in FIGURE 8.



```
xxxx=<>
yyyy= <
   zzz=<"wdifwbdfiwdufw">
   ww=<1992-04-12T12:01:00>
aaa= <fwefub>
bb=<"*J&h5g8biB9i8h">
ccc=<
   ddd=<
      eee=<124>
      fff=<"wdfiubwiefug">
   >
>
```

```
xxxx=<>
yyyy= <
zzz=<"wdifwbdfiwdufw">
ww=<1992-04-12T12:01:00>
aaa= <fwefub>
bb=<"*J&h5g8biB9i8h">
ccc=<
 ddd=<
eee=<124>
fff=<"wdfiubwiefug">
>
>
```

FIGURE 8 Digital signature check applied to knowledge artefact

9.3 Canonical Form – Archetype 'semantic view'

For hashing and signing to be useful, the input artefacts need to have two characteristics. Firstly, we need to know that the artefact has been validated, since there is no use in disseminating digitally authenticated but useless artefacts. Secondly, the effects of 'non-semantic' changes in the artefact must be removed. This requires a syntactic canonical form.

Both requirements can be achieved for archetypes and templates with a canonical form based on a 'semantic view' of an archetype, analagous to the 'interface class' idea in software development. The semantic view is created from a specific serialisation of the abstract syntax tree (AST) form of the artefact, which is its computable form. The full AST form is in fact defined by the *open*EHR AOM, but this contains all textual meta-data from the description, ontology and other sections of the archetype. The 'semantic' form of this model, suitable for generating a normalised serialisation for hashing has the following reduced form:

- the identifier;
- specialisation identifier, where present;

- concept code;
- definition section (comments stripped).

These objects would be represented in the same form as defined by the AOM. A suitable serialisation is the dADL syntax form. XML forms could be used, but they depend on which schema variant is in use, and there is no single normative *open*EHR XML-schema for the AOM.

TBD_6: canonical forms of other artefact types. Since all forms of archetypes and templates are now AOM-based (as of 1.5), a single canonical algorithm based on the AOM (with TOM extensions) can be described.

TBD_7: Operational template hashing & signing is required

END OF DOCUMENT