*open*EHR

Knowledge Artefact Identification

Issuer: openEHR Specification Program		
Revision: 0.5	*Pages:* 36	*Date of issue:* 18 Apr 2013
Status: DEVELOPMENT		

Keywords: EHR, health records, repository, governance

The *open*EHR Foundation is an independent, non-profit community, facilitating the sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Affiliates Australia, Brazil, Japan, New Zealand, Portugal, Sweden

Support **Issue tracker**: www.openehr.org/issues/browse/SPECPR
Web: www.openEHR.org

Amendment Record

Issue	Details	Who	Completed
0.5	Major update based on CKM clinical group analysis.	S Garde H Leslie I McNicoll T Beale	18 Apr 2013
0.2	Refinements to do with template identification. Review from Medical Centrum Alkmaar (Netherlands).	T Beale M van der Meer	01 Feb 2010
0.1	Initial Writing.	T Beale	09 Jul 2009

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.

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 `https://github.com/openEHR/specifications/blob/master/publishing/architecture/am/knowledge_id_system.pdf?raw=true`. New versions are announced on openehr-announce@openehr.org.

1.4 Changes to openEHR Release 1.0.2

The changes required to openEHR Release 1.0.2, and any reference models derived from or related to it, to enable the globalised identification system described here include:

Reference Model:

- augmented form of identifier class `ARCHETYPE_ID` to include organisational / package namespace, e.g.

  `org.openehr.ehr::openEHR-EHR-EVALUATION.problem.v1`
- `ARCHETYPED.archteype_id`, currently defined to have type `ARCHETYPE_ID` would be modified to be of type `List<ARCHETYPE_ID>` (`LOCATABLE.archetype_node_id` would be unchanged, and contain only at-codes to use the direct archetype id as currently done.

ADL/AOM 1.5 specifications:

- various modifications to `ARCHETYPE` class, to support version identifiers;
- the `concept` section of archetype identifiers (middle part of `ARCHETYPE_ID`) now relaxed to no longer require structure based on specialisation parents, e.g. ‘problem-diagnosis’ can now just be ‘diagnosis’, or any name preferred by designers.
- an optional `configuration` section is defined for operational archetypes and templates, allowing the exact list of source archetypes including version, revision and commit to be included.
- addition of a new `id_history` sub-section to archetype `description` section.

2 Introduction

2.1 The Problem

The notion of 'identification' for knowledge artefacts and models 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;
- define rules for expressing and resolving *references* between source artefacts, including version variants;
- define rules for identification of *compiled* / *operational* artefacts;
- define rules and lifecycle for *changing identifiers* (including version) of artefacts over time;
- define rules for identification when artefacts are retired, moved or 'forked'.

2.2 Ontological and Machine Identifiers

There are two general approaches to 'identification'. The first is the one used in software development: the 'ontological' human-readable identifiers. Under this approach, identifiers name an artefact (e.g. a source code file) 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 machine processable identifiers such as ISO Oids and GUIDs with accompanying de-referencing mechanisms. The two approaches are not mutually exclusive, nor are they equivalent.

An ontological identification scheme supports hierarchy, multi-dimensional concept spaces, flexible versioning, and formally reflects the artefact authors' and users' understanding of the concept space being modelled. Ontological identification supports many types of computational processing. A typical ontological software artefact identifier is `FastSortedList.java`. Within the software world,

[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

ontological style identifiers 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 ontological 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 ontological identifiers. Unlike ontological 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 ontological and machine identifiers are used. If machine identification only 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 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 an artefact such as an archetype 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 changed over time.
- a namespaced ontological, i.e. meaningful identifier for an artefact is *computed* from various properties of the artefact. For example, an archetype id can be computed from the properties `namespace`, `rm_class_name` and so on.
- the 'build' of an artefact (i.e. most recent version containing a change, no matter how small) can be identified in two ways:
    - a SHA1 hash on a canonical serialisation of the artefact
    - the `version` property from the ontological identifier properties.

This is a departure from the past, where no machine identifier has been assigned, and the artefact identifier was a static string, rather like a source file filename.

2.3 Artefacts, Meta-data and Registries

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 a '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 ontological identifier *properties*, and a machine identifier. Rather than being statically specified, ontological identifier(s) can be generated from specific properties of the artefact. For archetypes and templates, the relevant properties are defined on the ARCHETYPE 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.

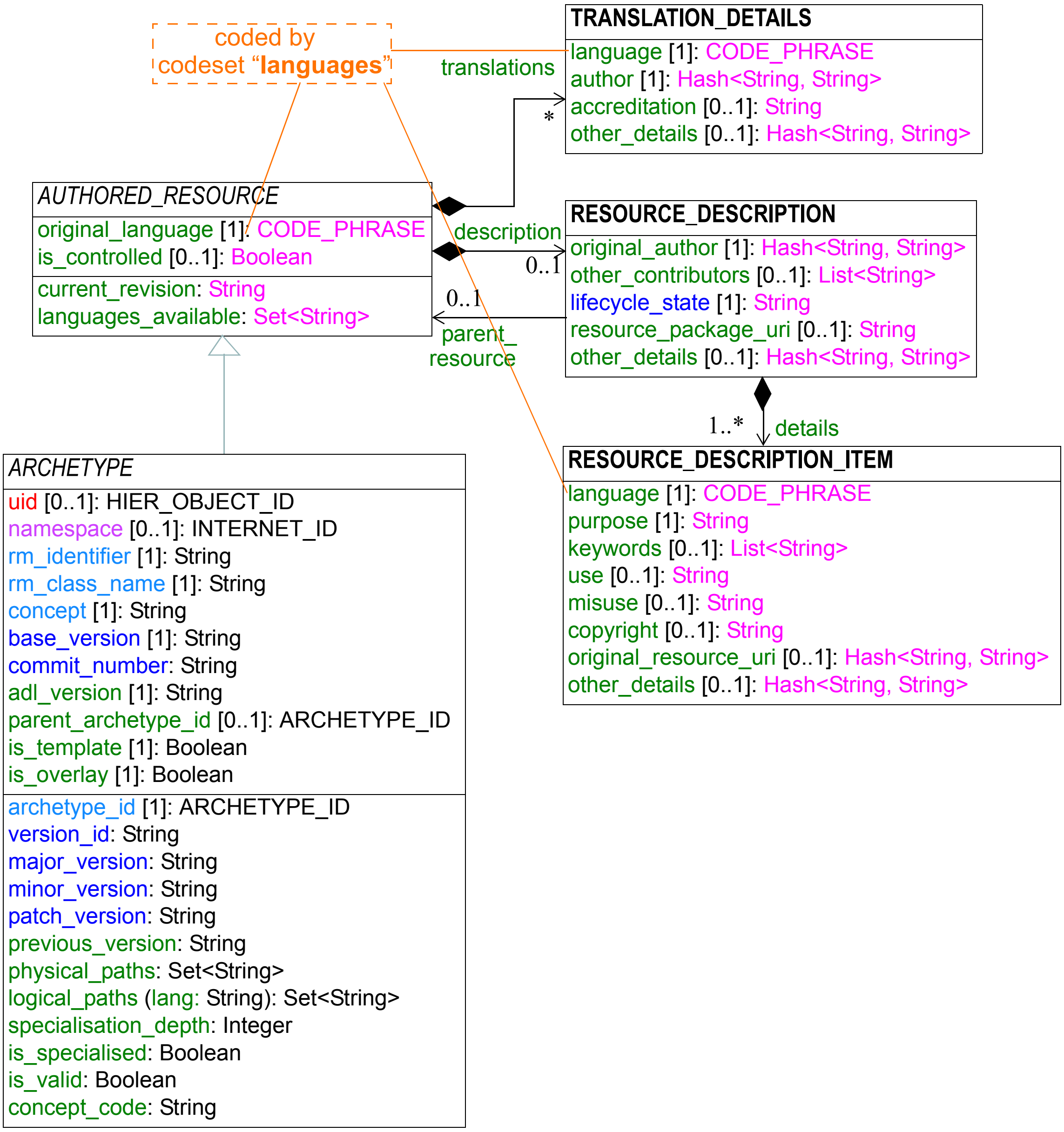

coded by
codeset "languages"
translations
*
TRANSLATION_DETAILS
language [1]: CODE_PHRASE
author [1]: Hash<String, String>
accreditation [0..1]: String
other_details [0..1]: Hash<String, String>
AUTHORED_RESOURCE
original_language [1]: CODE_PHRASE
is_controlled [0..1]: Boolean
current_revision: String
languages_available: Set<String>
description
0..1
0..1
parent_
resource
RESOURCE_DESCRIPTION
original_author [1]: Hash<String, String>
other_contributors [0..1]: List<String>
lifecycle_state [1]: String
resource_package_uri [0..1]: String
other_details [0..1]: Hash<String, String>
1..*
details
RESOURCE_DESCRIPTION_ITEM
language [1]: CODE_PHRASE
purpose [1]: String
keywords [0..1]: List<String>
use [0..1]: String
misuse [0..1]: String
copyright [0..1]: String
original_resource_uri [0..1]: Hash<String, String>
other_details [0..1]: Hash<String, String>
ARCHETYPE
uid [0..1]: HIER_OBJECT_ID
namespace [0..1]: INTERNET_ID
rm_identifier [1]: String
rm_class_name [1]: String
concept [1]: String
base_version [1]: String
commit_number: String
adl_version [1]: String
parent_archetype_id [0..1]: ARCHETYPE_ID
is_template [1]: Boolean
is_overlay [1]: Boolean
archetype_id [1]: ARCHETYPE_ID
version_id: String
major_version: String
minor_version: String
patch_version: String
previous_version: String
physical_paths: Set<String>
logical_paths (lang: String): Set<String>
specialisation_depth: Integer
is_specialised: Boolean
is_valid: Boolean
concept_code: String

FIGURE 1 Core Archetype Classes

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

There are three distinct groups of properties shown in FIGURE 1 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_identifier`, `rm_class_name`, `concept` form the basis of the main part of an ontological identifier, e.g. `openEHR-EHR-OBSERVATION.bp_measurement`;
- various properties supporting versioning:
  - `base_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 commi 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' (unstable) parts of non-release version ids.

Additionally, functions such as `archetype_id` and `version_id` are defined to return respectively the full archetype identifier, and the full version string (computed from `base_version`, `commit_number` and `description.lifecycle_state`). The functions `major_version`, `minor_version` and `patch_version` extract the various parts of the 3-part `base_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 ontological identifier 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.

3.2 Formal Model

Both *managed* and *unmanaged* knowledge artefacts are distinguished. Their identifiers are defined as follows:

```
artefact_id: managed_artefact_id | unmanaged_artefact_id
managed_artefact_id: namespace '::' ontological_id
unmanaged_artefact_id: ontological_id
namespace: publisher_org_id { '.' package_id }*

publisher_org_id: V_REVERSE_DOMAIN_NAME
library_id: V_ALPHANUMERIC_NAME
package_id: V_ALPHANUMERIC_NAME

V_ALPHANUMERIC_NAME: [a-zA-Z][a-zA-Z0-9_]+
V_REVERSE_DOMAIN_NAME: See IETF RFCs 1035, 123, and 2181.
```

3.2.1 Namespace

Note that the `namespace` is constructed from a publisher organisation identifier plus zero or more packages, in the same manner as software library identifiers.

All archetypes and templates should be identified with this style of identifier. Any archetype or template missing the `namespace` part is assumed to be an unmanaged artefact.

An optional system of hierarchical package identifiers can be added to the namespace identifier defined above, in order to enable each artefact management organisation to distinguish between 'libraries' or packages of artefacts.

Examples:

`org.openehr.`**`ehr`** - EHR archetypes library at *open*EHR.org
`org.openehr.`**`demographic`** - demographics library at *open*EHR.org
`uk.nhs.`**`maternity`** - maternity library at UK NHS
`edu.nci.`**`cancer_studies`** - cancer studies library at US National Cancer Institute

3.2.2 Ontological Identifier

The ontological identifier is defined as follows, assuming a non-exhaustive list of four types of artefact:

```
ontological_id: ontological_root '.' major_version
ontological_root: archetype_id_root
            | template_id_root
            | subset_id_root
            | query_id_root
major_version:      'v' {V_NUMBER}+

V_NUMBER: [0-9]+
```

3.2.2.1 Archetype Identifier

The existing *open*EHR `ARCHETYPE_ID` is understood as the ontological identifier of an archetype within the concept space in which it was authored. The same kind of identifier is used forADL/AOM 1.5 templates. For both archetypes and templates, the final part of the identifier has historically been known as the 'version'. In the scheme defined here, it is the major version. Its presence in the archetype identifier means that a new major version of an artefact is actually a distinct artefact. Therefore, when we speak of an 'artefact', we are speaking of the *major version* of a notional artefact.

The archetype ontological identifier is defined by the following grammar, which is a slightly simplified version of the grammar for the `ARCHETYPE_ID` type defined in the *open*EHR Support IM specification:

```
archetype_id_root: qualified_rm_class_name '.' domain_concept
qualified_rm_class_name: rm_publisher '-' rm_closure '-' rm_class_name
rm_publisher:       V_ALPHANUMERIC_NAME
rm_closure:         V_ALPHANUMERIC_NAME
rm_class_name:      V_ALPHANUMERIC_NAME
domain_concept:     V_SEGMENTED_ALPHANUMERIC_NAME

V_ALPHANUMERIC_NAME: -- see above
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: name of class or equivalent entity in the reference model on which the artefact is based;

domain_concept: the domain concept name;

version: numeric version identifier.

> **NB:** this is a simplified but backwardly compatible form of the grammar in Release 1.0.2 *open*EHR, Support IM specification.

The essence of this definition is a two-dimensional identifier consisting of reference model class and domain concept, with added versioning.

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

```
openEHR-EHR-EVALUATION
```

This historically has been used in *open*EHR to identify the reference model class on which an archetype is based. It includes the publisher of the reference model (e.g. “*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.

3.2.2.2 Domain Concept

The second part of the ontological identifier is a domain concept identifier. Domain concept identifiers have historically been natural language words or phrases, typically shortened to a mnemonic form, e.g. “bp_measurement” in the archetype identifier `openEHR-EHR-OBSERVATION.bp_measurement.v1`. Also historically, this 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.

Legacy Semantics

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 would still be allowed, but would no longer have 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, including the use of namespaces.

Ontological Basis

The more important aspect of the domain concept identifier, from an ontological point of view, is its origin. 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 domain 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 the managing organisation of the artefact.

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*. More specifically, it is assumed that specific nodes within the ontology are ultimately related to nodes that correspond to the generic information categories of a notional information (i.e. 'reference') model. This leads to an ontology of the form shown below.

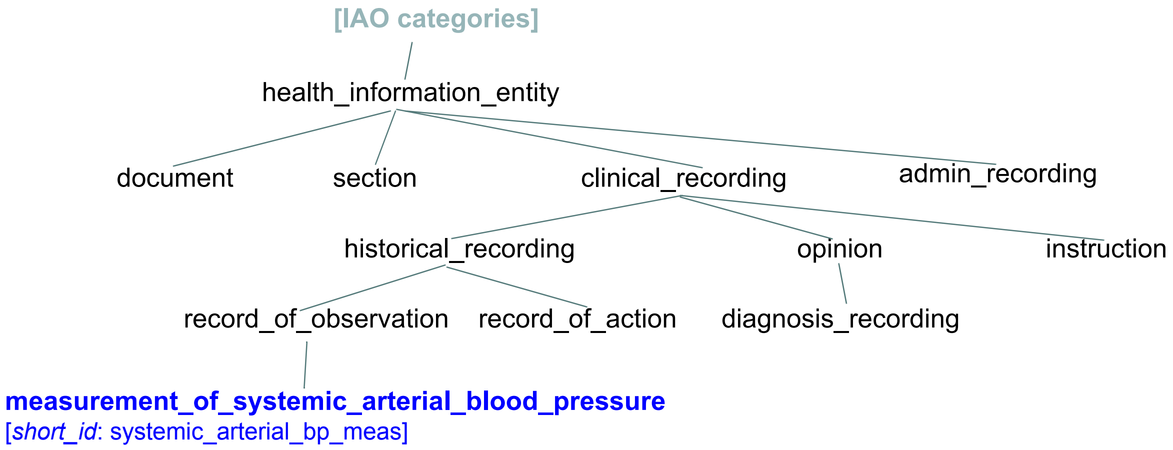


FIGURE 2 Notional archetype concept ontology within a namespace

In this ontology, the black entities are categories that define the common notions of 'document', 'section', and various kinds of 'clinical recording' (aka 'clinical statement', 'entry' etc in various published health data information models). 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), but this is not a pre-requisite for establishing this ontology.

The blue node defines the category corresponding to a 'systemic arterial blood pressure observation'. 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 long names may be deemed as 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 / OWL) and that ontology identifiers are checked to ensure uniqueness.

Facilities to manage such ontologies should be located within artefact repositories, so that every added archetype, template or subset was assigned a short concept name 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 concept names can be proposed as the short id for an ontology class defined for the archetype. If there is a clash, a new archetype concept short id will be needed, and the archetype will need to be republished under a different identifier.

3.2.3 Template Identifier

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

```
template_id_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, both in *open*EHR and in the ICT industry in general. 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_id_root: qualified_terminology_id '.' domain_concept
qualified_terminology_id: terminology_originator '-' terminology_name
terminology_originator: V_REVERSE_DOMAIN_NAME
terminology_name: V_ALPHANUMERIC_NAME
```

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
```

TBD_1: The most obvious 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 howerver.

3.2.5 Query Identifier

There has been little experience with identification of queries as a designed 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_2: ontological id for queries

3.3 File System Recommendations

Archetypes and templates can be represented on the file system as normal text files. How such files are identified depends on the intended use of the files. The base rule in this specification is that nothing is assumed about filenames. It is assumed that tools will always inspect the contents of artefact files to determine their identifiers and other details.

Nevertheless, for purposes of practical tool building, sharing of artefacts and general sanity of developers, it is likely to be useful for filenames to follow some simple guidelines, particularly in non-production environments, as follows:

- name an artefact file after the non-namespaced artefact ontological identifier;
- use a *type extension* to distinguish the various source and operational forms of the same artefact, in the usual manner of file naming.

The recommended extensions are as follows:

- .adl - ADL 1.4 syntax archetype
- .adls - ADL 1.5 syntax archetype
- .adlf - ADL 1.5 syntax flattened archetype
- .xaom - XML serialised form of AOM archetype
- .opt - XML operational template

3.4 Versioning

3.4.1 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.

3.4.2 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.

In the above, the ‘formal definition’ refers to the definition, ontology/term_definitions and ontology/constraint_definitions sections of an archetype or template, and the technical contents of a terminology ref-set - in other words, the parts of the artefact other than information meta-data.

Lexically, the version identifier is defined as follows:

```
version_id: release_version [extension]
release_version: major_version '.' minor_version '.' patch_version
extension: modifier commit_number
modifier: '-rc' | '+u'
```

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 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 ‘unstable’, 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 per-artifact 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 *ontological* 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.

3.4.3 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 thes 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**. 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 5.

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.

3.4.4 Lifecycle Model

As with software, knowledge artefacts follow a lifecycle with identified states, representable by a state diagram. A traversal through the state diagram, shown in FIGURE 3, corresponds to the development of changes to an artefact leading to a release of a specific version.

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 accepted practice to choose 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 numeric version id, appended with '+uC' where C is the current commit number.
- 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'.

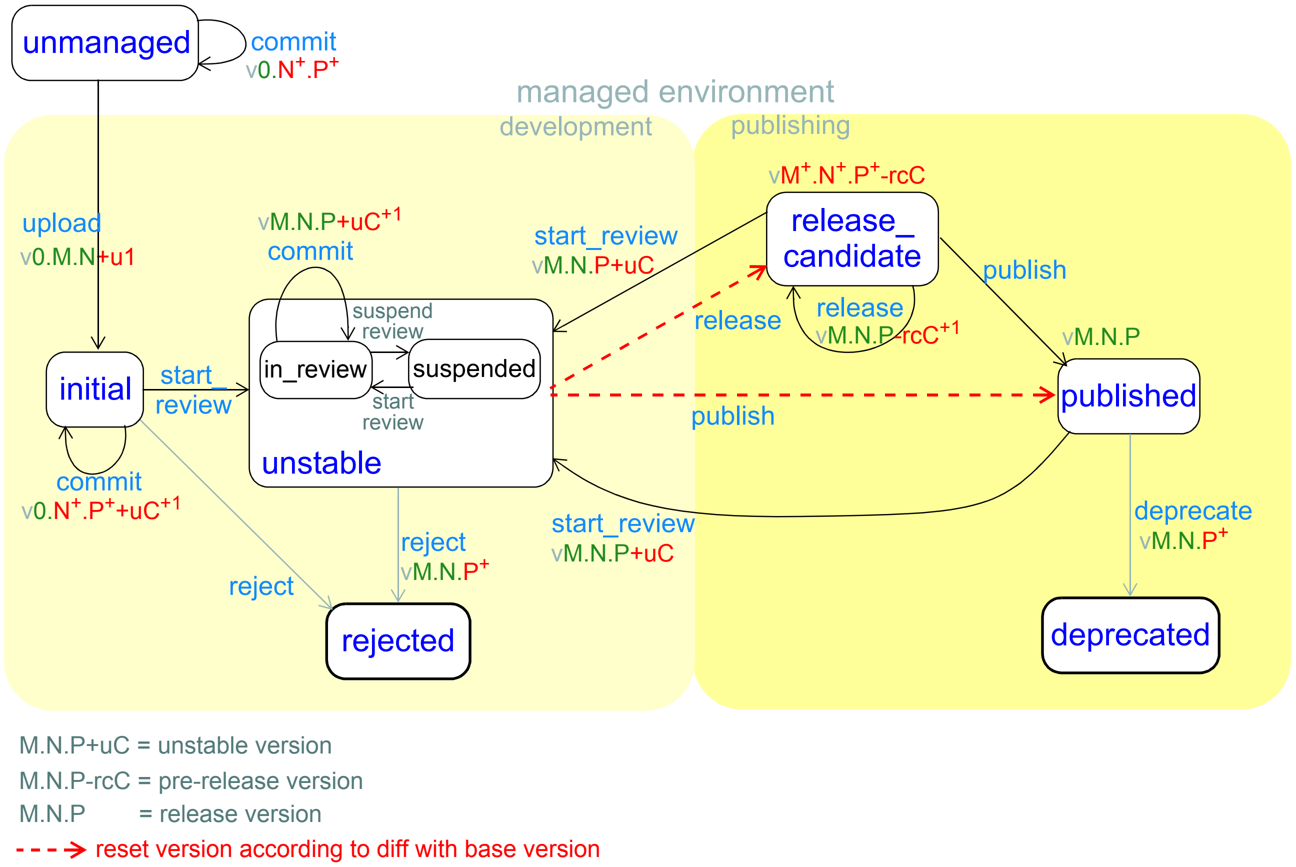


FIGURE 3 Development Lifecycle and Versioning

- 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:
  - published into a pre-release cycle, with a version of the form N.M.P-rcC, which allows community testing and further review. It may also be release from the release_candidate state, with a stable version id of the form N.M.P.
  - released directly to a stable version of the form N.M.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 ‘rejected’.

According to the basic 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.

4 Distributed Governance

4.1 Overview

This section deals with how knowldege artefact identifiers are managed in a distributed environment consisting of unmanaged artefact authors as well as management organisations. 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. The environment is illustrated below.

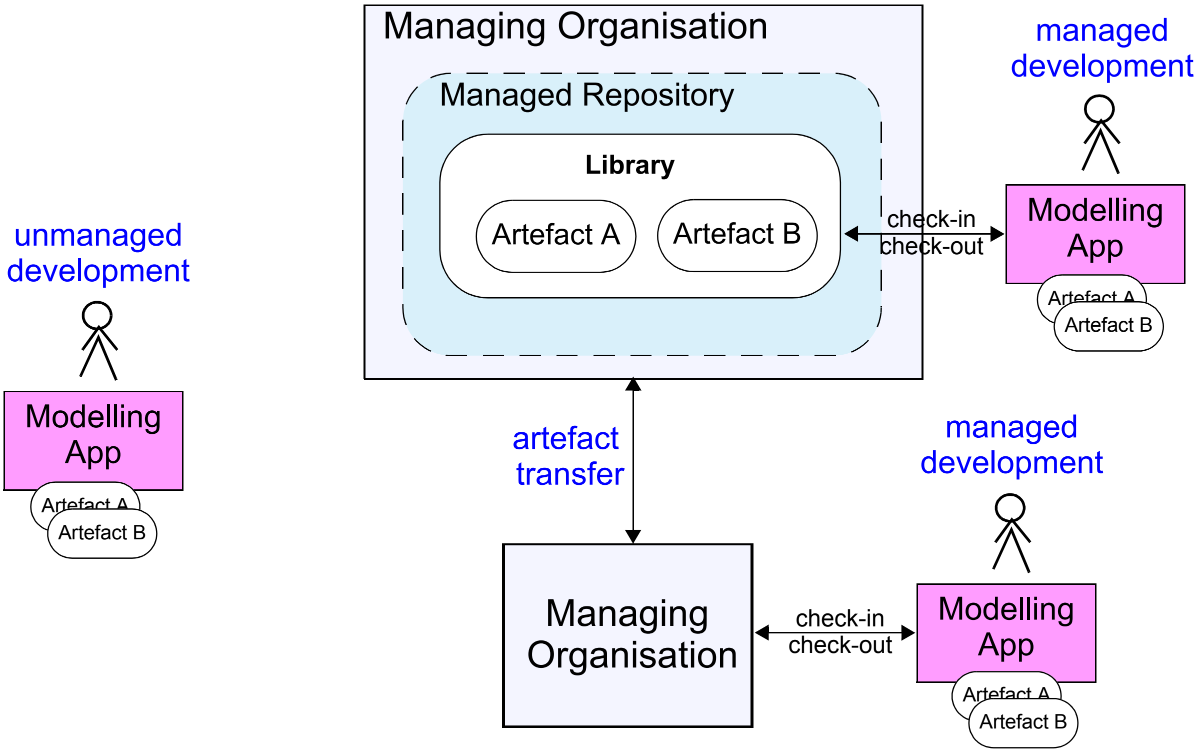


FIGURE 4 Distributed Development Environment

4.2 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 ontological identifier is changed to the namespaced form, with the reverse domain name of the organisation acting as the namespace;
- it is assigned a newly generated Guid as its uid;
- if its version number is higher than v0.N.P, it is reset to v0.0.1, otherwise it is left unchanged;
- various meta-data items are set, including copyright.

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

4.3 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 managing organisation. 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 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 managing organisation will most likely adopt them for use. The question is: how are the new releases of the artefact identified?

With respect to the ontological identifier, two basic strategies are available: retain the original ontological identifier, or change it to reflect the new managing organisation. An argument against changing it is that identifier continuity would be preserved, ensuring that archetype references in 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 indentified 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 managing organisation 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 managing organisation, 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 for purposes 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 ontological identifier and the machine identifier. With respect to the ontological identifier, any of the following are assumed to be mutable:

- *namespace*: at a minimum this will always change;
- *concept*: the artefact concept identifier, derived from an ontology 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 ontological 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.

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 new identifier and the old. 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.

5 Referencing

This section describes how artefact are referenced, by other artefacts and software. The general principal for referencing is that ontological references 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 ontological identifiers or machine identifiers.

5.1 Source Artefact References

This section describes the scenarios and representation required for identifier-based referencing between artefacts.

5.1.1 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 ontological identifier, as shown in the following example.

```
ACTIVITY[at0001] ∈ {-- 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. A namespace would be included to express a reference to an archetype outside the current namespace.

A reference such as the above will allow the following actual archetypes at runtime:

- any *released* version variant of openEHR-EHR-ITEM_TREE.medication.v1, i.e. any minor version such as v1.0.4, v1.2.49 etc;
- any 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;
- any child archetype of openEHR-EHR-ITEM_TREE.medication.v1, i.e. archetype whose 'specialise' clause references this archetype.

TBD_3: Currently, the last point is how the AOM/ADL 1.5 spec defines the semantics, following normal OO substitutability semantics. An alternative would be to allow the use of the Snomed compositional syntax << operator to indicate children being allowed.

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. 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 the same as its ontological identifier, with the optional addition of a variable amount of finer grained version information.

5.1.2 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 ontological identifier. At some point it will often be the case that the minor version must be included as well. As noted above, this is because minor versions of archetypes and templates can include structural addition, and therefore affect the structure of the final document / data-set etc. It may even be the case that patch levle 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 outgoing references 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 seems reasonable to allow '-rc' and '+u' variants as well.

5.1.3 Between Specialised Archetypes

A specialised archetype refers to its parent using the ontological form of the identifier, including only the major version. Two possible variants can occur:

- With a non-namespaced identifier. 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 revision 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.

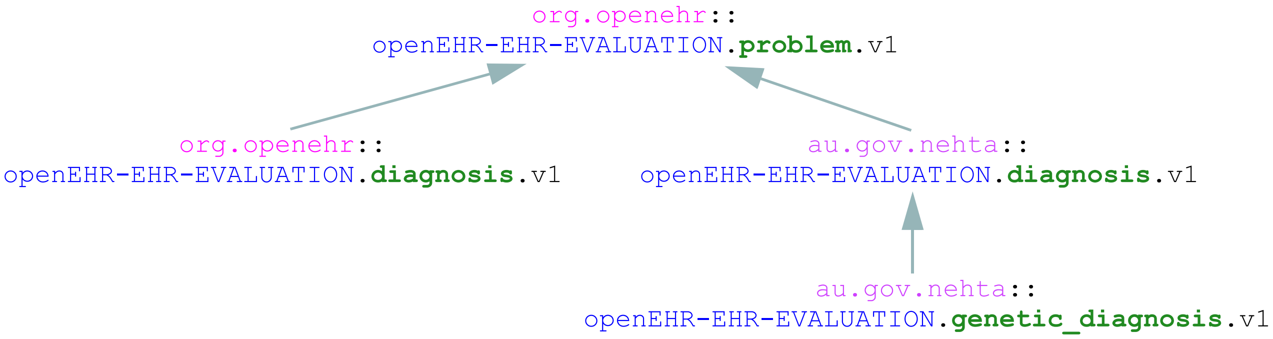

org.openehr::
openEHR-EHR-EVALUATION.problem.v1
org.openehr::
openEHR-EHR-EVALUATION.diagnosis.v1
au.gov.nehta::
openEHR-EHR-EVALUATION.diagnosis.v1
au.gov.nehta::
openEHR-EHR-EVALUATION.genetic_diagnosis.v1

FIGURE 5 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.

5.1.4 Formal Model

An artefact reference is defined as follows:

```
artefact_ref: ontological_root '.' version_ref
version_ref: 'v' major_version ['.' minor_version ['.' patch_version
             [extension]]]
```

5.2 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.

5.2.1 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[at0020] 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.

5.2.2 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 [at0004.1] occurrences ∈ {0..1} ∈ {
    include ∈ {True}
       archetype_id ∈ {
```

```
                ARCHETYPE_ID ∈ {
                    namespace ∈ {...}
                    concept ∈ {<< investigation_methodology OR
                                << investigation_protocol}
                    ...
                }
            }
        }
```

The above kind of referencing relies on a solid underpinning for the concept part of the ontological identifier, such as an ontology basis.

TBD_4: No formal model for this type of slot has yet been defined.

5.3 Query Sets

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 things 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. Archetype-based queries contain archetype identifiers and paths, and may contain template identifiers and paths.

In typical usage, archetypes are referenced by their ontological identifier, i.e. including major version number only. Because of the way specialisation is defined in the archetype formalism, a query referencing an archetype major version will match:

- any data points from specialisation children based on this major version, which are redefinitions on *paths existing in the original parent*;
- any data points from other minor versions of this major version, which are redefinitions on *paths existing in the original major version*.

Since a new minor version can introduce new paths, as long as they are backwardly compatible with existing paths, there may be data paths that can only be addressed with respect to a particular minor version of an archetype. This can mean in some circumstances that the minor version of archetype identifiers needs to be included in the references in the queries.

The general case is that any level of vesion referencing is assumed to be valid, even though production systems will almost always be limited to using major version references.

TBD_5: term & ref-set references - ensure that terminology ids (e.g. Oid, non-oid etc) can be matched.

5.4 Operational Artefacts

Operational artefacts such as flattened archetypes and operational templates generated by the compiler tools are built from source artefacts, including by reference resolution from within some source artefacts to others within the current configuration of the local and imported Libraries. 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 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”>
    >
>
```

5.5 References from Data

5.5.1 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.

5.5.2 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'.

5.5.3 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
```

5.5.4 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: ontological_root '.' version_id_ref
version_id_ref: 'v' version_id
```

5.5.5 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.

5.5.5.1 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.

5.5.5.2 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.

6 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:

7 Scenarios

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

7.1 Minor Version Upgrade

To Be Continued:

7.2 Major Version Upgrade

To Be Continued:

7.3 Templates using Archteypes and Subsets

To Be Continued:

7.4 Artefact Transfer / Fork

To Be Continued:

8 Artefact Authentication

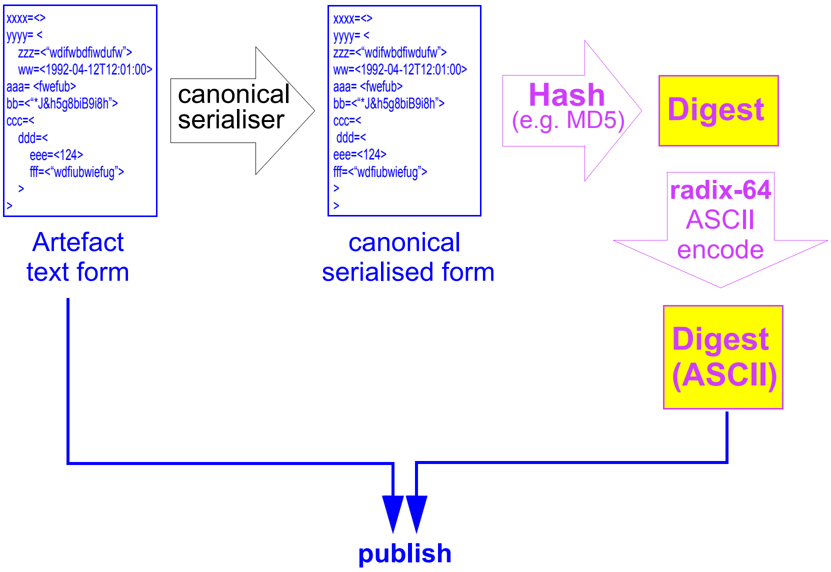
In theory, revision information should be reliable, and no two physical knowledge artefacts should exist that are either identical but have different identifiers and/or revision information, or are different but are identified as being the same. However, in practical systems, such situations can occur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

TBD_6: hashing on source v operational artefacts? Consider templates that don't change but referenced archetypes that do.

8.1 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, regardless of revision information. 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.



```
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 6 Integrity check applied to knowledge artefact

8.2 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 Publishing Organisation of an artefact. The PKI approach is for each PO to generate a key pair, and to provide the public key to the Central Governance Authority. Signing is then carried out using the PO private key on the hash digest already generated for an artefact. The modified process is illustrated in FIGURE 7.

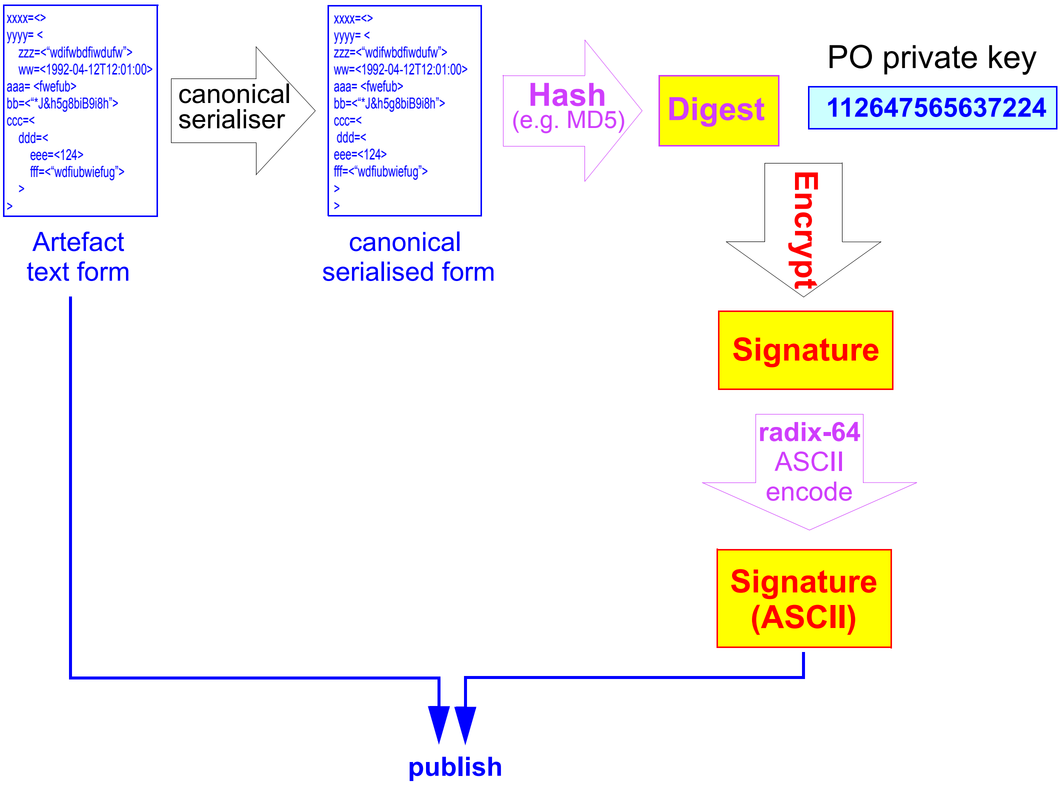


FIGURE 7 Digital signature check applied to knowledge artefact

8.3 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_7: 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_8: Operational template hashing & signing is required

END OF DOCUMENT