Release 1.0.2

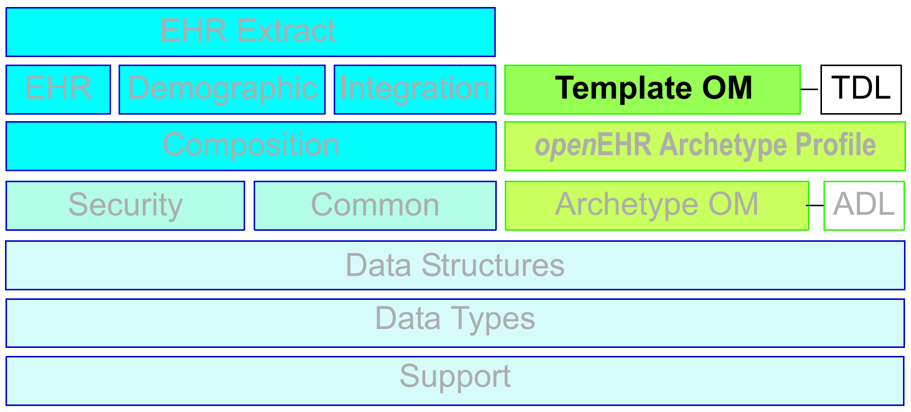
The *open*EHR Archetype Model

*open*EHR Templates

Editors: {T Beale, S Heard}[a]		
Revision: 1.0	*Pages:* 37	*Date of issue:* 20 Apr 2009
Status: DEVELOPMENT		

a. Ocean Informatics

Keywords: EHR, ADL, health records, archetypes, templates



Copyright Notice

Amendment Record

Issue	Details	Raiser	Completed
R E L E A S E 1.0.2 candidate			
1.0	**SPEC-178**. Add template object model to AM.	T Beale	20 Apr 2009
R E L E A S E 1.0.1			
0.5	Minor content modifications.	T Beale	13 Mar 2007
R E L E A S E 1.0			
0.5rc1	**CR-000178**. Add Template Object Model to AM. Initial Writing	T Beale	10 Nov 2005
R E L E A S E 0.96			

Trademarks

Acknowledgements

The work reported in this document was funded by Ocean Informatics and the UK National Health Service (NHS).

Table of Contents

1 Introduction

1.1 Purpose

This document describes three related formalisms used for defining *open*EHR templates:

- Template Definition Language (TDL) - an abstract language for expressing *template defin-intions* in a syntactic fashion;
- The Template Object Model (TOM) - an object model that expresses the same semantics as TDL in a structural fashion, in the same way as the AOM structurally expresses the semantics of the Archetype Definition Language (ADL);
- The Operational Template Model (OTM) - an object model describing the standalone, *operational template* which is generated from template definitions and referenced archetypes and terminologies.

TDL provides a computer-processible and human-readable syntax in which to write and persist templates, while the model can be used as a basis for building software that processes archetypes and templates, independent of their persistent representation. As with archetypes, a faithful XML serialisation is also available. The Template Object Model is the normative expression of the semantics of *open*EHR templates. Many users and tool producers will implement only the TOM and an XML serialisation of templates.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations using *open*EHR;
- Academic groups using *open*EHR;
- Medical informaticians and clinicians interested in health information;
- Health data managers.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Architecture Overview

Related documents include:

- The *open*EHR Archetype Definition Language (ADL)
- The *open*EHR Archetype Object Model (AOM)

1.3 Nomenclature

In this document, the term ‘attribute’ denotes any stored property of a type defined in an object model, including primitive attributes and any kind of relationship such as an association or aggregation. XML ‘attributes’ are always referred to explicitly as ‘XML attributes’.

The term ‘template’ used on its own always means an *open*EHR template definition, i.e. an instance of the model described in this document. An operational template is always denoted by its full name in *open*EHR.

1.4 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/svn/specification/TRUNK/publishing/architecture/am/tom.pdf. New versions are announced on openehr-announce@openehr.org.

Blue text indicates sections under active development.

1.5 Tools

To Be Continued:

2 Overview

2.1 Context

Templates constitute a third layer above archetypes and the reference model in the *open*EHR application architecture shown in FIGURE 1, and provide the means of defining groupings of archetype-defined data points for particular business purposes. They support bindings to terminology subsets specific to their intended use, and can be used to generate or partly generate a number of other artefact types including screen forms and message schemas.

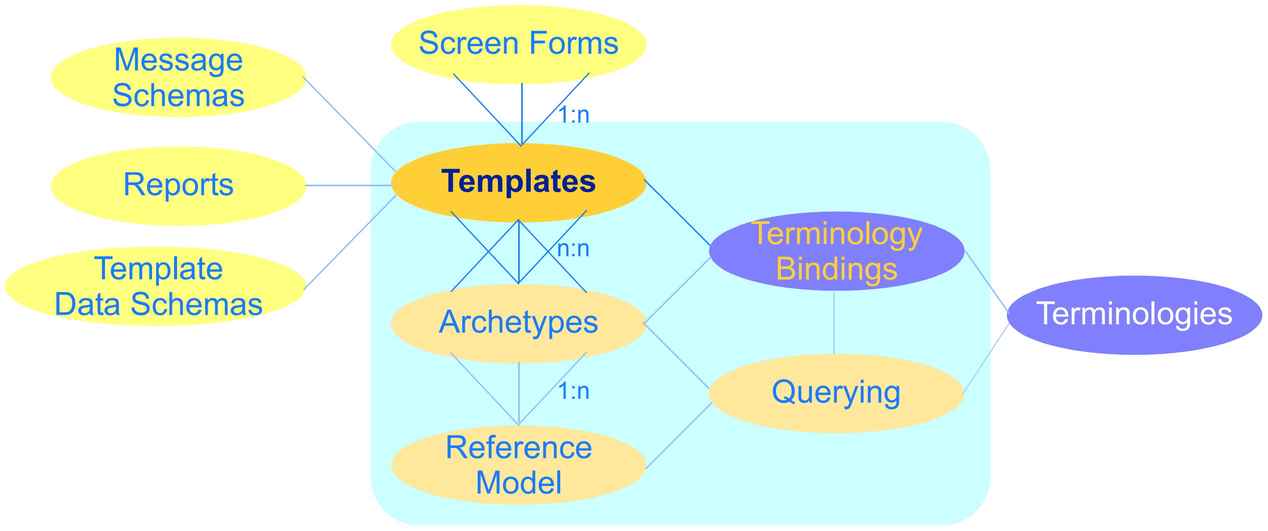


FIGURE 1 The *open*EHR Semantic Architecture

Templates are defined by the following specifications:

- the Archeype Object Model (AOM);
- this specification: the Template Object Model (TOM) and the Template Definition Language (TDL).

A related artefact is the *open*EHR *palette*, which which defines the local language and terminology preferences on a per-archetype basis, also defined in this specification.

2.2 Basic Semantics

2.2.1 Purpose

An *open*EHR template is an artefact that enables the content defined in archetypes to be used for a particular use case, i.e. business event. In health this is normally a 'health service event' such as a particular kind of encounter between a patient and a provider. Archetypes define content on the basis of topic or theme e.g. blood pressure, physical exam, report, independently of particular business events. Templates provide the way of using a particular set of archetypes, choosing a particular (often quite limited) set of nodes from each and then limiting values and/or terminology in a way specific to a particular kind of event, such as 'diabetic patient admission', 'ED discharge' and so on. Such events in an ICT environment nearly always have a logical 'form' (which may have one or more 'pages' or subforms and some workflow logic) associated with them; as a result, an *open*EHR template is often a direct precursor to a form in the presentation layer of application software. Templates are the technical means of using archetypes in runtime systems.

This job of a template is as follows:

- aggregate archetypes into larger structures by indicating which archetypes should fill the slots of higher-level archetypes;
- remove unnecessary archetype elements (‘data points’);
- narrow existing constraints;
- set default values.

A template may aggregate any number of archetypes, but choose very few data points from each, thus having the effect of defining a small data set from a very large number of data points defined in the original archetypes.

2.2.2 Formal Definition

Formally, an *open*EHR template is a source artefact, and is expressed differentially with respect to other models, in this case, archetypes and the underlying reference model. This has the practical effect that its contents are mainly references to archetypes, with associated constraint redefinitions and default values. There may also be constraints on reference model elements not mentioned in the referenced archetypes. When a template is successfully compiled, it produces an operational template that can be thought of as the ‘flat’ form of a template, in the same way as the flat form of a specialised archetype is the resulting archetype due to compressing the inheritance of a lineage of specialised archetypes.

The actual semantics of the constraints expressed in a template are very similar to those in a specialised archetype, with some differences. The semantics can be summarised as follows.

- Templates do not have a specialisation level of their own - they are a set of constraints on a group of archetypes, each of which may have different specialisatoin levels.
- Constraints are defined in the form of a path within an archetype and particular constraints that apply at that path - as for specialised archetypes.
- Only paths that are further constrained by the template are mentioned in the template definition - the constraints at all other paths in referenced archetypes remain intact (i.e. can be considered to be ‘inherited’, just as with specialised archetypes), similarly for parts of the reference model not constrained by the archetypes or the template.
- Default values can be defined in a template, and are defined with respect to specific archetype paths.
- Unlike specialised archetypes, templates cannot redefine object node identifiers (at-codes).
- However, object nodes within container attributes can be *cloned*.

These semantics ensure templates can not change the semantics of archetypes or introduce new semantics (i.e. the ontology of an archetype cannot be changed by a template). Accordingly, *all data created due to the use of template are guaranteed to conform to the referenced archetypes and underlying reference model*.

The formal definition of *open*EHR templates is mostly provided by the Archetype Object Model, with a few additional semantics defined in the Template Object Model. It is also provided in a syntax form called Template Definition Language (TDL), which like the TOM, is a small addition to ADL.

2.3 The Role of TDL

The Template Definition Language is used in this document as a normative human-readable and computable syntax specification for authoring templates, and it fully expresses their semantics. It may also be used by tool developers for testing and operational purposes. However, the normative definition of template semantics is given by the Template Object Model, for which other serialised or syntax formats can be specified. An XML format is specified by *open*EHR, currently as a W3C XML Schema (XSD). Experience with archetypes and XML shows that this is likely to be improved upon in the future with more efficient versions of the schema, and there may also be other alternatives defined such as in Schematron. As a consequence there may be no single normative XML format for expressing templates. The only reponisibility of tool builders is to ensure that any *open*EHR template tool implements the semantics of the TOM and at least one recognised serial format.

2.4 Computational Environment

FIGURE 2 shows the relationships among the various template artefacts and related archetype artefacts. A template (top-left) in its document (i.e. serialised form) is defined as a set of constraint statements similar to a specialised archetype. It refers to one or more archetypes and usually imposes further constraints. A template parser converts a template into an in-memory object form described by the Template Object Model. The document form template may be expressed in TDL, or in any XML or other equivalent derived from the Template Object Model. An operational template builder generates an in-memory Operational Template which is a standalone artefact (contains all relevant parts of its template, referenced archetypes and terminology) that can be used to generate other computational artefacts including screen forms. The builder takes a palette as an input, which has the effect of removing other languages and terminologies when generating the operational template, ensuring that each resulting artefact only contains what is needed for local use.

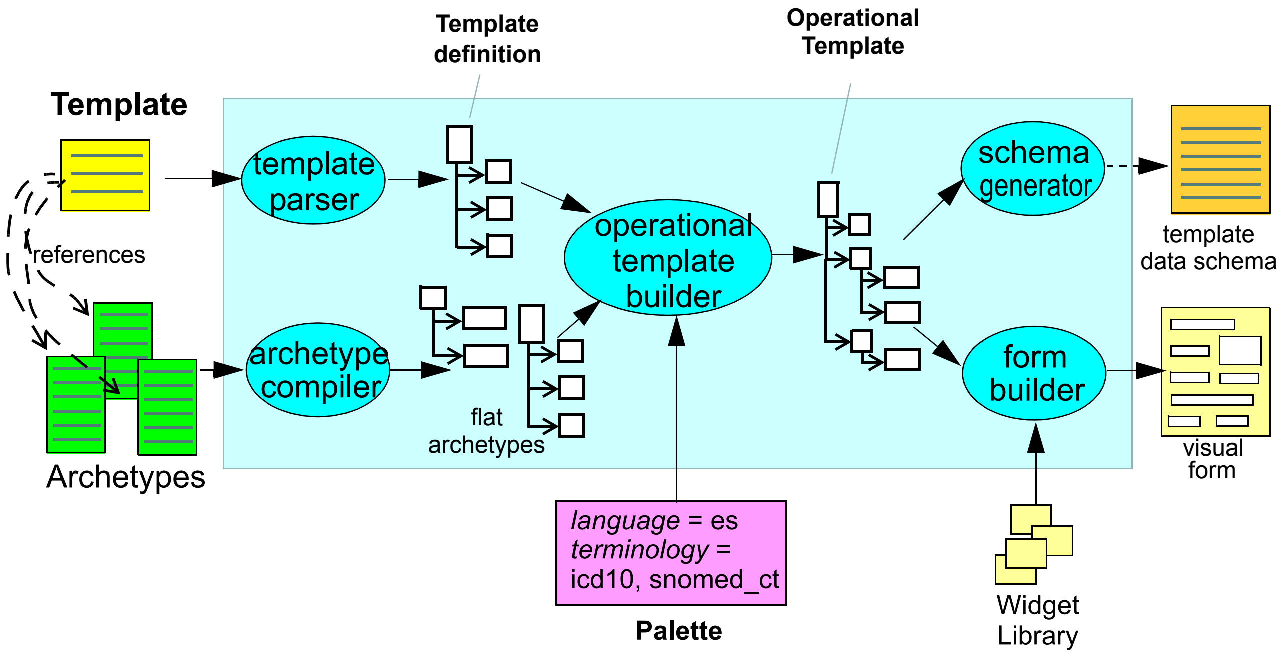


FIGURE 2 Template Tool Chain

3 Requirements

3.1 Overview

Templates need to be able to perform four main functions as follows.

Aggregation: the first is to combine archetypes into larger structures by filling archetype slots with archetypes or previously defined templates.

Element choice: the second is to choose which parts of the chosen archetypes should remain in the final structure, by removing unneeded elements, and indeed whole parts of structures specified in an archetype or the underlying reference model (where not constrained by the archetypes being used)

The third is to optionally add further constraints to the chosen archetypes or reference model, specific to the needs of the template. This may include narrowing of value ranges, and refined constraints on terminology.

The fourth function is to define default values of leaf or near-leaf structures. Default values will usually be used to pre-populate fields on a screen form, or within a template-based message.

A further general requirement on templates is of a documentary nature: how are they described, and how are they linked to existing resources to do with defining data, records and user interfaces.

The following sections describe the requirements for *open*EHR templates. Most of the requirements described here have been determined by real world use of prototype models of *open*EHR templates in health.

3.2 Identification

A globally unique identification system is required such that templates can be authored locally without reference to a central repository or identification authority, and yet still be distinguished if they are shared, or if data created by them are merged.

A human-readable structural identifier is also needed so that templates can be referred to by human authors, and template references within templates can be understood by humans.

A single kind of identifier may be used to satisfy the above purposes, or multiple identifiers may be needed.

Template versions must be included in the identification system in such a way that multiple versions of a given template can be distinguished. The versioning system should support at least 3-part version identification.

3.3 Documentary Requirements

3.3.1 Descriptive Meta-data

A similar set of meta-data as supported by archetypes should also be supported by templates, including items such as:

- purpose;
- use;
- misuse;

- audit details of creation and modification;
- references to resources used in the authoring process;
- copyright status.

Descriptive meta-data should be limited to items that directly help explain the template, rather than cross-references to every possible related item in an enterprise, which should be dealt with by other means.

3.3.2 Author Annotations

It should be possible for a template author via the use of a tool to create annotations per node of the template definition structure. Note that such annotations only apply to nodes mentioned in the template, i.e. nodes further constrained in the template. At each node, it should be possible to create one or more annotations, each with a unique tag with respect to other annotations on the same node. Annotation content is required to be normal text.

Annotations should be modifiable over time by authors. It is expected that many annotations will be meaningful only during the authoring period, but not at the time of release. A way is needed to easily remove or ignore some or all node-level annotations.

3.3.3 Institutional Meta-data and Links

There is a need to be able to create links between new artefacts like archetypes and templates, and institutional or external resources, typically documentary or design artefacts about the same topic expressed in natural language or older technologies. Within the template building environment the need is to be able to assert and persist links between such resources and templates, or parts of templates, which requires a means of referencing individual nodes in a template. It is not expected that templates themselves need to support such linking internally, rather they would be represented in an artefact external to templates.

3.4 User Interface Requirements

TBD_1: template to ‘form’ relationship

TBD_2: visibility of nodes on a form

3.5 Semantic Requirements

3.5.1 Relationship to Archetypes

A template is required to constrain one ‘root’ archetype, including any constraint redefinition and slot-filling possible within the root archetype.

3.5.2 Slot Filling

The most basic requirement is to be able to define larger structures composed from archetypes, by choosing filler archetypes for the slots defined in ‘higher’ (in the aggregation sense) archetypes. There are three possibilities that have to be supported:

- completely specifying the fillers for a slot;
- specifying some fillers for a slot, where remaining slot fillers may be decided at runtime;
- leaving the slot as defined by the archetype, leaving all slot fillers to be decided at runtime.

In addition, existing templates should be allowed to be used in a slot in place of an archetype. Experience with prototype templates in *open*EHR has shown the need to be able to define a template for a subtree of content that will always be constrained in the same way in all top-level objects in which it is used in the institution. Rather than redefine the same constraints in each top-level template, it is clearly preferable to be able to reuse a single definition of such content.

Typical examples are the Oximetry, Blood Pressure and Heartrate templates of the *open*EHR `OBSERVATION` class, which are reused in a number of `SECTION` and `COMPOSITION` templates.

Nesting needs to work in such a way that if the referenced template is changed, it is changed in all templates using it, i.e. a true reference. It also needs to function such a way that the referencing template can define additional constraints over and above what the referenced section defines.

3.5.3 Constraint Refinement

Templates should be able to refine the constraints defined in any included archetypes, or create new constraints on the parts of the reference model relevant to those archetypes, in the same was as for archetypes, with the following exception: template constraints cannot change existing archetype node identifiers (unlike specialised archetypes).

In particular, terminology constraints need to be redefinable as in specialised archetypes, i.e. any of the following:

- redefine open constraint on text to be set of coded terms;
- narrow coded term constraint to a subset;
- redefine an external subset (defined using an ac-code in an archetype) to a narrower external subset;
- redefine an external subset to an internal subset.

3.5.4 Default Values

Template authors need to be able to define default values for nodes defined in the template, or in an underlying archetype, even if the template does not change the node in question, and similarly for elements in the reference model for which there may or may not be constraints in the archetypes or template. A 'default' value is one which will be usually be displayed and will be used at runtime unless the user or application actively supplies something different.

Experience has shown that defaults need to be settable on primitive leaf types, i.e. String, Integer, Real, Boolean and the date/time types. These usually occur within subtypes of the *open*EHR type `DATA_VALUE` (e.g. `DV_QUANTITY` and `DV_CODED_TEXT`).

3.5.5 Conditional Constraints

In some cases it is needed to be able to state a constraint that is applied only if a condition is true. A typical situation is to make the existence of a subtree dependent on the existence or runtime values of data found in other subtrees. A condition is an expression combining variables and constants with boolean, arithmetic and relational operators, to generate a boolean result that can be used to decide whether to apply a constraint.

The following types of condition variables need to be supported:

- *external variable*: variables such as 'today's date';
- *EHR variable*: variables such as 'date of birth', 'gender' from the EHR;

- *existence of another path*: the existence of a path in the data, which implies that at runtime an optional node was included;
- *value on other path*: the value of a leaf object in the data on a particular path.

Constants that can be used in condition expressions include the primitive types, and dates and times.

Conditions should include the facility for a natural language description of the condition so that the intent is recorded.

3.6 Optimisations

In some cases it is desirable to be able to pre-populate some terminology subsets, particularly those that are small, change rarely, or for situations where the relevant terminologies will not be available in the deployment environment at runtime. For large subsets, e.g. the set corresponding to Snomed-ct 'bacteria' (numbering in the 10s of thousands of terms), prepopulation is not desirable.

It should be possible to state within a template whether a particular subset of a specific terminology should be pre-populated.

TBD_3: if the subset was defined in the archetype, we really want to specify that a particular *binding* should be prepopulated, but if it was specified in the template, then we are choosing the binding directly?

For subsets that would normally have internal structure intact at runtime, the pre-populated version should also retain this structure, allowing the subset to be treated at runtime in the same way as if it had been available from an online terminology service. For example, a Snomed-CT subset that contains IS-A links should retain these when pre-populated.

Pre-population of templates is effected in operational templates. Consequently, when a terminology is updated, affected operational templates need to be regenerated. This requires a way of recording which templates need to be accessed in order to regenerate the corresponding operational template when changes occur in terminologies.

3.7 Tooling Requirements

TBD_4: Nested templates in a different colour

TBD_5: cut and paste of a template, template section

4 Template Definition Language (TDL)

4.1 Introduction

An slightly extended form of the ADL syntax known as the Template Definition Language (TDL), is used as an abstract syntax for *open*EHR templates. As a superset of ADL, TDL enables the re-use of existing ADL software in building template tools. The extensions concern semantics specific to templates, and are done in a way compatible with the existing ADL syntax.

The structure of a TDL template is the same as an ADL archetype, with the sole exception that the first ssection is called 'template' rather than archetype.

```
template (...)
    template_id
language
    dADL language description section
description
    dADL meta-data section
definition
    cADL structural section
[rules
    rules]
[annotations
    dADL section]
[revision_history
    dADL section]
```

4.2 Basics

4.2.1 Relationship to Archetypes

A template has a single archetype as its root object, so may be said to be a template of a single reference model concept in the same way as the root archetype.

4.2.2 Node Identification

A template cannot redefine object node identifiers found within archetypes it uses. In order to maintain the basic rule of runtime node sibling level uniqueness, renaming or cloning must be used, as described in section 4.5.4 and section 4.5.5.

4.3 Header Sections

4.3.1 Template Identification Section

This section introduces the template and must include an identifier. A typical `template` section is as follows:

```
template (tdl_version=1.0)
    uk.nhs.cfh::openehr-EHR-COMPOSITION.admission_ed.v1
    e65542c8-881e-48c4-9f4a-75deaeaf3be8
```

Two types of identifiers are supported for templates, enabling both human and computer use. The *template unique identifier* is a defined as a Guid, allowing it to immediately be generated by tooling without reference to a central repository. Each version of a template that is made available for use is assigned a new Guid, ensuring that versions of the same logical template are never confused.

A second structural identifier is defined by the class `TEMPLATE_ID` (`rm.support.identification` package). This is a multi-axial identifier of the same form as an archetype identifier.

The structural identifier can be created by any organisation that has a registered domain name, and allows related versions of the same template to be associated without reference to any external index.

4.3.2 Language Section and Language Translation

The `language` section of a template is identical to that of an archetype.

4.4 Description

The description section of a template is of the same form as that of an archetype. The semantics are defined by the `AUTHORED_RESOURCE` class in the `rm.common.resource` package.

4.5 Definition

4.5.1 Overview

The definition section of a template is expressed in a slightly extended form of the cADL syntax used in archetypes. The extension enables archetype references to be stated at the root of the definition section and, along with template references, within archetype slots. The `use_archetype` and `use_template` keywords achieve this respectively.

A definition section always starts with a reference model type assocated with an archetype identifier (rather than a node identifier, as in an archetype), indicating the root archetype of the template. In its most minimal form, no further constraints are stated, and the definition section would appear as follows:

```
definition
    EVALUATION[openEHR-EHR-EVALUATION.problem.v1]
```

More typically, some further constraints will be stated on the root archetype, and the definition section will resemble the following, where the ellipsis represents constraints on the archetype as used within the template:

```
definition
    EVALUATION[openEHR-EHR-EVALUATION.problem.v1] ∈ {
        ...
    }
```

4.5.2 Slot Filling

Slots defined in archetypes take the form of constraints on the identities of archetypes that can be used at a ‘joining point’ within another archetype. Specifying how the slots will be filled is one of the primary jobs of a template. Within a template, there are three possibilities for dealing with a slot defined in a referenced archetype.

- Leave the slot untouched, enabling archetype(s) to be chosen at runtime, as occurs with some kinds of very dynamic data.
- Specify one or more *slot-fillers* matching the slot constraint to fill the slot, being any mixture of:
  - archetypes that match the constraints expressed in the slot;
  - other templates whose root archetype matches the constraints expressed in the slot.

- Specify one or more slot fillers as above, but leave the slot open for further choices of slot-fillers at runtime.

The first case corresponds to the situation where choice of archetypes can only be done at runtime. Slots left open in this fashion cause the template to be assigned 'open' status. When the choice is finally made at runtime to fill remaining open slots, the template is altered accordingly (even if only in an in-memory form) and 'closed'. It can then be used to generate an operational template. Nothing needs to be stated in a template to leave a slot untouched.

For the second case, if an archetype is specified to fill the slot, further constraints on the archetype *as used within the slot* can be stated. Specifying a template creates an *embedded template*, and only its identifier is stated, as no further redefinition is allowed.

The following example shows a slot taken from a SECTION archetype for the concept 'history_medical_surgical' archetype.

```
SECTION[at0001] occurrences ∈ {0..1} ∈ {-- Past history
    items cardinality ∈ {0..*; unordered} ∈ {
        allow_archetype EVALUATION [at0002] ∈ {-- Past problems
            include
                archetype_id/value ∈ {
                    /openEHR-EHR-EVALUATION\.clinical_synopsis\.v1
                        |openEHR-EHR-EVALUATION\.excluded(-[a-z0-9_]+)*\.v1
                        |openEHR-EHR-EVALUATION\.injury\.v1
                        |openEHR-EHR-EVALUATION\.problem(-[a-z0-9_]+)*\.v1/}
            exclude
                archetype_id/value ∈ {/.*/}
        }
    }
}
```

This slot specification allows EVALUATION archetypes for the concepts 'clinical synopsis', various kinds of 'exclusions' and 'problems', and 'injury' to be used, and no others. The following fragment of TDL shows how the slot is filled in a template without stating any further constraints. In this syntax, the usual at-code semantic markers have been replaced by the identifiers of archetypes or templates designated to fill the slots.

```
use_archetype SECTION[openEHR-EHR-SECTION.history_medical_surgical.v1] ∈ {
    /items ∈ {
        use_archetype EVALUATION[at0002 = openEHR-EHR-EVALUATION.problem.v1]
        use_template EVALUATION[at0002 = uk.nhs.cfh::openEHR-EHR-
                                          EVALUATION.ed-diagnosis.v1]
        use_archetype EVALUATION[at0002 = openEHR-EHR-EVALUATION.
                                                      clin_synopsis.v1]
    }
}
```

In addition to specifying slot fillers, it is possible to specify that the slot is 'closed', i.e. not available for further filling at runtime. This is done by including an overridden version of the archetype slot object itself, with the 'closed' constraint set, as in the following example:

```
use_archetype SECTION[openEHR-EHR-SECTION.history_medical_surgical.v1] ∈ {
    /items ∈ {
        use_archetype EVALUATION[at0002 = openEHR-EHR-EVALUATION.problem.v1]
        allow_archetype EVALUATION[at0002] closed
    }
}
```

Slots can be recursively filled in the above fashion, according to the possibilities offered by the chosen archetypes. The following TDL fragment shows two levels of slot-filling:

```
To Be Continued:        FIND A REAL EXAMPLE OF THE FOLLOWING:
     use_archetype COMPOSITION[openEHR-EHR-COMPOSITION.xxx.v1] ∈ {
         /content ∈ {
            use_archetype SECTION[at0001=openEHR-EHR-SECTION.yyy.v1] ∈ {
               /items ∈ {
                  use_template EVALUATION[at0002=uk.nhs.cfh::
                                             openEHR-EHR-EVALUATION.xx.v1]
                  use_archetype EVALUATION[at0002=openEHR-EHR-EVALUATION.xx.v1]
                  use_archetype EVALUATION[at0003=openEHR-EHR-EVALUATION.xx.v1]
               }
            }
         }
     }
```

4.5.3 Template Constraints

In the case of archetypes used to fill slots, or the root archetype, further constraints can be stated with respect to the original archetype, in a similar fashion to the definition of a specialised archetype.

However, within a template there are some differences, as follows:

- new object node identifiers cannot be created, either by redefinition or extension;
- any object can be renamed;
- a default value can be given for leaf and near-leaf objects;
- within container attributes:
    - ‘cloned’ nodes can be created from an existing node identifier, but with unique values for the ‘name’ attribute inherited from the *open*EHR LOCATABLE class.

The following example shows constraints applied to two archetypes used to fill the slot from the history_medical_surgical SECTION archetype from above.

```
     use_archetype SECTION[openEHR-EHR-SECTION.history_medical_surgical.v1] ∈ {
         /items ∈ {
            use_archetype EVALUATION[at0001=openEHR-EHR-EVALUATION.problem.v1] ∈
            {
               /data/items ∈ {
                  CLUSTER[at0026] occurrences ∈ {0} -- remove ‘Related probs’
                  ELEMENT[at0031] occurrences ∈ {0} -- remove ‘Age at res’n’
               }
            }
            use_archetype EVALUATION[at0001=openEHR-EHR-EVALUATION.
                                                   clin_synopsis.v1] ∈
            {
               /data/items ∈ {
                  ELEMENT[at0003] ∈ {
                      ...
                  }
               }
            }
         }
     }
To Be Continued:
```

4.5.4 Object Renaming

TBD_6: this section to be removed or rewritten

Most classes in the *open*EHR reference model inherit from the class LOCATABLE, which defines the *name* and *archetype_node_id* attributes. The latter is used to record a copy of the at-code from the node in the archetype used to create the corresponding node in the data, while the former is a runtime name value, required to distinguish sibling data nodes of the same container attribute. In data, the *name* attribute by default would typically be set to the text value of the at-code identifying the node, in the language of the locale. For container child nodes having occurrences greater than 1, and for which multiples are actually generated in the data, the *name* attribute values must be made unique, typically by appending counters or some similar method.

Archetypes may constrain the *name* attribute to be something other than the default, although this is not common, since naming tends to be locally determined. Templates on the other hand are more likely to be used to set the name value of data items. This is achieved using an extension of the object identifier syntax used in archetypes, in which a second field corresponding to the desired name value is added. The following example is from an antenatal_check SECTION archetype, where the *name* of the at0003 (“Evaluation”) node in the archetype is constrained to the word “Assessment”.

```
/items ∈ {
    ELEMENT[at0003, “Assessment”]
}
```

Since the name attribute in LOCATABLE is of type DV_TEXT, it can also be set to an instance of DV_CODED_TEXT, i.e. a coded term, rather than simply free text. This is achieved with the following syntax.

```
/items ∈ {
    ELEMENT[at0003, snomed_ct::386053000|Assessment|]
}
```

A multiply occurring object within a container can be renamed in the same way. In data created at runtime, the uniqueness requirement will force each instance of such a node to carry the name value modified to be unique in the same way as if the default name value had been used.

TBD_7: in any renaming, occurrences must be set to max = 1, since can’t have multiple items with same name (uniqueness rule).

4.5.5 Object Cloning

Container atttributes occur frequently in reference models. An archetype may or may not define particular variants of child objects within the container. If it does, each will have a different at-code. A template also provides the possibility to define variants of a given container child object, distinguished by name rather than node identifier.

This is done by ‘cloning’, which means creating copies of the object node as defined in the archetype, and then for the original and all copies, constraining the name value such that all siblings thus created have unique names with respect to each other. The `clone` keyword is used for this purpose. The following example shows how this is done to create a templated version of a generic ‘laboratory-result’ archetype to represent ‘thyroid test result’.

```
/data/events[at0002]/data/items ∈ {
    clone ELEMENT[at0013, snomed_ct::65428006|TSH|]
    clone ELEMENT[at0013, snomed_ct::259356007|Free T3|]
    ....
}
```

Further constraints may appear on renamed and cloned nodes, as per the following example.

```
    /data/events[at0002]/data/items ∈ {
        clone ELEMENT[at0013, snomed_ct::65428006|TSH|] ∈ {
            value ∈ {
                C_DV_QUANTITY <...>
            }
        }
        clone ELEMENT[at0013, snomed_ct::259356007|Free T3|] ∈ {
            value ∈ {
                C_DV_QUANTITY <...>
            }
        }
        ....
    }
```

The following example shows three clones of the ‘any event’ node from an archetype such as the *open*EHR archetype for ‘body temperature’.

```
    /data/events ∈ {
        clone EVENT[at0003, "1 minute"] -- clone of 'any event' node
        clone EVENT[at0003, "3 minute"] -- clone of 'any event' node
        clone EVENT[at0003, "6 minute"] -- clone of 'any event' node
        ....
    }
```

The result of this is that three data nodes are defined for use in data in addition to the original ‘any event’ node defined by the archetype. However, a combination of a rename and two clones could have been used to limit the possibilities in data to only three nodes as follows:

```
    /data/events ∈ {
        EVENT[at0003, "1 minute"] -- rename of 'any event' node
        clone EVENT[at0003, "3 minute"] -- clone of 'any event' node
        clone EVENT[at0003, "6 minute"] -- clone of 'any event' node
        ....
    }
```

In this example, the first item is a redefinition by renaming of the archetyped node, whereas in the previous example, all three nodes are separate clones distinct from the original node.

4.5.6 Default Values

Default values are available in templates to support the situation where only one value is possible for a data item due to the specific nature of the template. For example, a blood pressure archetype may allow a number of possible values for ‘patient position’, such as ‘lying’, and ‘sitting’, ‘standing’. When used in a hospital, the patient will usually be lying so a default value for this can be set, as shown in the following example:

```
    /data/events[at0006]/state/items[at0008]/value ∈ {
        DV_CODED_TEXT
        default (DV_CODED_TEXT) <
            defining_code = <[snomed::163033001]> -- lying blood pressure
        >
    }
```

Default values are expressed in dADL syntax, since they are instances of objects, rather than being constraints. The example above only sets the default value, but it could have also modified the constraint on the value object as well, as in the following version (where the standing blood pressure possibility from the archetype has been removed):

```
    /data/events[at0006]/state/items[at0008]/value ∈ {
        DV_CODED_TEXT ∈ {
            defining_code ∈ {
                [snomed::163033001, -- lying blood pressure
                163035008] -- sitting blood pressure
            }
        }
        default (DV_CODED_TEXT) <
            defining_code = <[snomed::163033001]> -- lying blood pressure
        >
    }
```

Default values can be set in the same way on container objects, such that one or more container objects distinguished by node identifier or name (if renaming has been used in the template) within the same container can have a default value assigned to them.

To Be Continued: example

A default value is either of the same type as specified by the corresponding archetype node (*rm_type_name* attribute) or any subtype allowed by the reference model.

The `default` keyword is used to introduce a dADL block expressing the value of an instance. It appears directly after the object block to which it applies; if no template additions have been made to this block, only the type name and node identifier, where defined, from the archetype are mentioned.

4.5.7 Conditions

A common need in templates is to be able to make a constraint dependent upon a runtime value, either of data in another part of the same template, elsewhere in the same record, or an external variable. Any constraint block can be made dependent on a condition, expressed in the ADL assertion language. Conditions are expressed using the normal ADL rule statements. The `exists` operator is used to qualify paths that must exist depending on some condition, as in the following example:

```
    rules
        $date_of_birth:ISO8601_DATE ::= query("patient", "date_of_birth")
        $current_date - $date_of_birth < P2Y implies exists /path/to/infant/data
```

In the following example, the existence of the path /x/y/z is dependent upon the presence of the /context path.

```
    exists /context implies exists /x/y/z
```

In the following example, the existence of the path /x/y/z is dependent upon the start_time of an encounter being before 1994.

```
    /context/start_time < 01-01-1994 implies exists /x/y/z
```

4.5.8 Terminology Binding

To Be Continued: can include direct binding to terminology subset on any DV_CODED_TEXT

4.6 Annotations

Annotations are defined as a separate list of items, each keyed by template path. This allows them to be used during template design, but easily ignored, reduced or removed from a template (including. in serialised forms such as XML) at the time of publishing. By the time the template is released for use, the annotations should only include content that is germane to the final form of the template, rather than temporary design notes.

Annotations are included in a template in the same way as for ADL archetype, e.g.

```
To Be Continued:        example
```

5 The template Package

5.1 Overview

The *open*EHR `am` package is illustrated in FIGURE 3, showing the `template` package in relation to the other `am` packages.

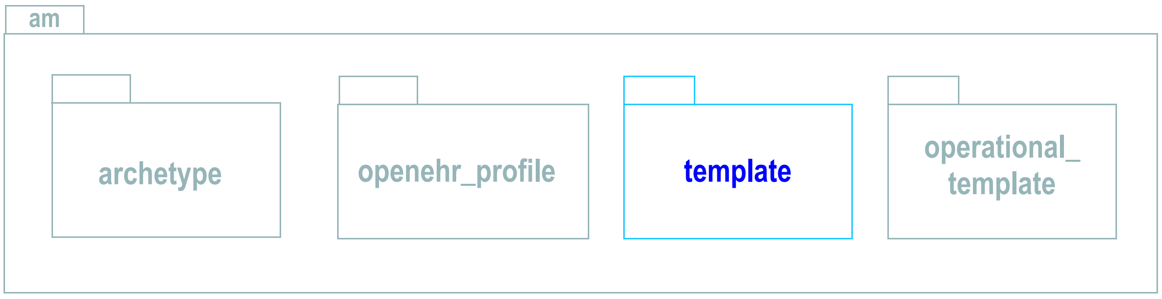


FIGURE 3 openehr.am.template Package in context

The `template` package defines the definition, or 'source' form of an *open*EHR template and is shown in FIGURE 4.

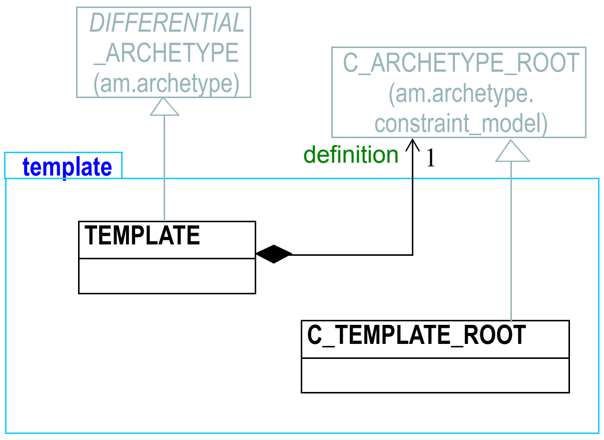


FIGURE 4 openehr.am.template package

The model has been defined using only two new classes in addition to the *open*EHR Archetype Object Model (AOM), enabling the existing semantics of the AOM to be reused. All of the required template semantics are available within the AOM. FIGURE 5 illustrates the template top-level object structure. Mandatory parts are shown with a bold association.

The sections below describe how this object model supports the semantics required in templates.

5.2 Design

A template source definition is an instance of the `TEMPLATE` class, which is defined as a specialisation of `DIFFERENTIAL_ARCHETYPE`. The only difference in a template is that its *definition* root object is specialised to the type `C_ARCHETYPE_ROOT`, a specialisation of the `C_COMPLEX_OBJECT`

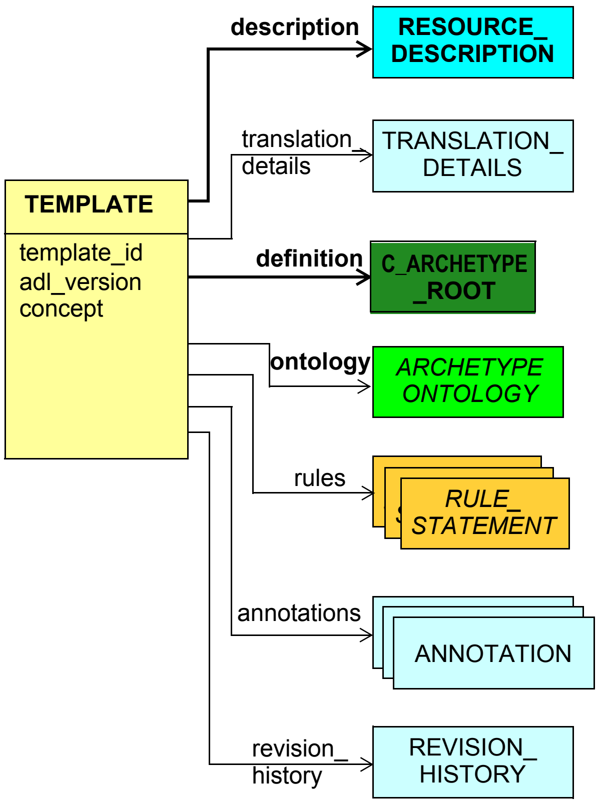


FIGURE 5 Template top-level object structure

class in which the *node_id* attribute is an archetype identifier rather than a normal internal node identifier.

5.2.1 Template Identification

Identification of templates is the same as for archetypes. There are two possible identifiers: the *archetype_id*, the structured multi-axial form defined by the `ARCHEYTPE_ID` class, and a Guid form, provided by the inherited *uid* attribute. The structural identifier is for human readability and use in ontological classifications.

If used, the Guid identifier can be immediately generated by tooling without reference to a central identification service.

TBD_8: Each version of a template that is made available for use is assigned a new Guid, ensuring that versions of the same logical template are never confused.

5.2.2 Meta-data

Templates support the same resource-level and node-level meta-data as archetypes, defined by the `openehr.rm.common.resource` package.

5.2.3 Definition section

The definition part of a template is expressed as a hierarchy of *`C_OBJECT`* descendants and `C_ATTRIBUTE` objects, in the same was as an archetype. In a template, the definition structure has two functions:

1. to define archetype slot fillers;

2. optionally to define constraint refinements on the archetypes designated as slot-fillers.

The first function is achieved using the ARCHETYPE_SLOT and C_ARCHETYPE_ROOT classes, while the second is done in a similar way to redefinition in a specialised archetype.

Further details of slot-filling and constraint redefinition are given below.

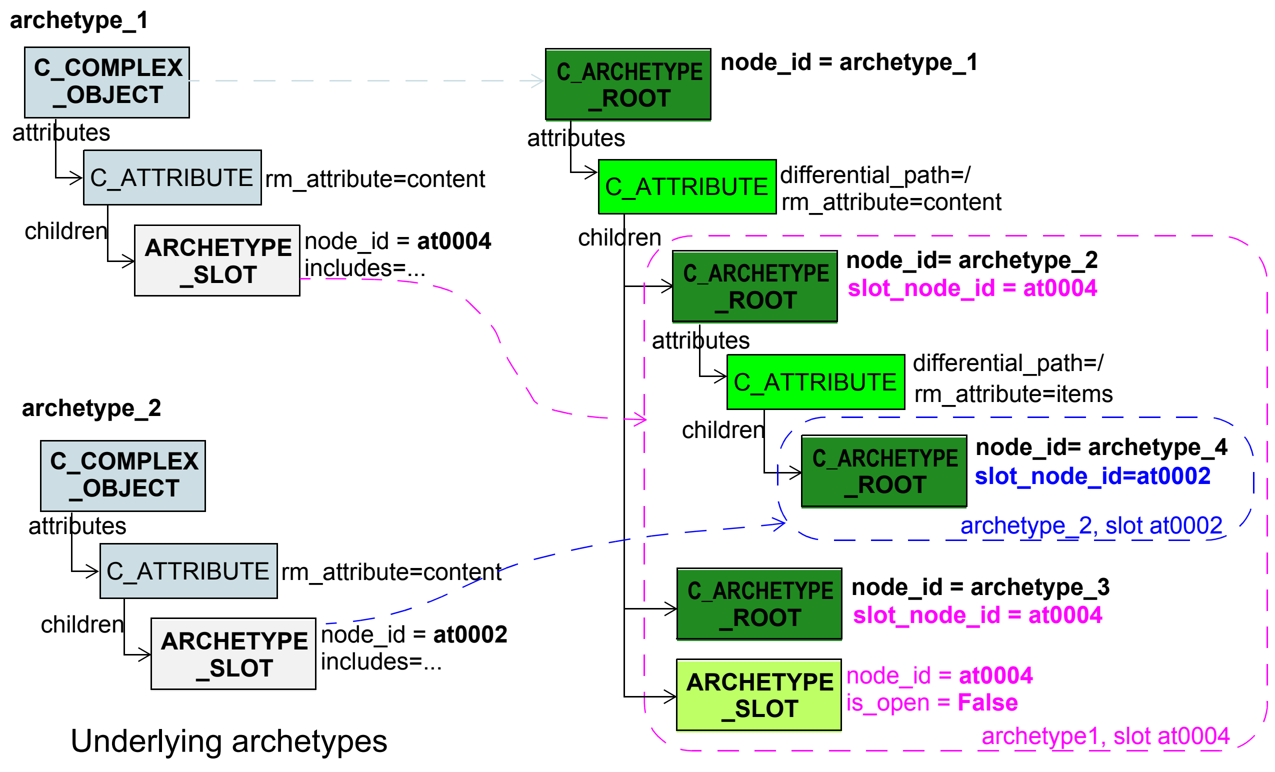


FIGURE 6 Typical template definition object structure

5.2.3.1 Slot-filling

The template definition root object is always a C_ARCHETYPE_ROOT object, designating the flat archetype which is being templated. Where there are slots within the archetype, the template can do two things:

- fill the slots with appropriate archetypes;
- ‘close’ the slot for further filling at runtime.

Slot filling is accomplished in the template by the presence of C_ARCHETYPE_ROOT nodes under a C_ATTRIBUTE node representing the attribute and path of an ARCHETYPE_SLOT within an archetype. Each such node has its *slot_node_id* attribute set to the node identifier of the slot in its original archetype.

The second function, closing a slot, is achieved by including an ARCHETYPE_SLOT node within the template, which overrides the *is_open* attribute to False. If a slot is left open when the operational template is generated, it should be made available at runtime for filling by the user.

Due to the use of slot fillers and slot closing, there are three variations on slot-filling that can occur in a template:

- it may *completely* specify the archetypes filling a slot (the slot is designated closed);
- it may specify only some slot fillers and leave the slot open;

- it may make no statements about a slot, defining neither slot fillers nor the closing of the slot.

In the second two variants, the runtime system will be able to add further archetypes to a slot (these will be flattened archetypes, as required for runtime deployment).

The objects defined in a template for both slot filling and closing can be thought of as 'replacing' the object in the underlying archetype (i.e. an `ARCHETYPE_SLOT`) in just the same way as any differential overriding is performed in a specialised archetype.

Slot fillers (ie. `C_ARCHETYPE_ROOT` instances) obey the normal existence, cardinality and occurrences constraints, i.e.:

- they satisfy the existence constraint, and where relevant, the cardinality of the containing attribute;
- they satisfy the occurrences of the original slot definition;
- they may further constrain the above occurrences to a narrower value by specifying their own occurrences overrides.

5.2.3.2 Template Inclusion

Each slot filler can be either an archetype, or another template. Templates are included in the same way as archetypes, except that an instance of `C_TEMPLATE_ROOT` is used rather than `C_ARCHETYPE_ROOT`. When a template is included, its content is not further constrainable by the referencing template. This is indicated by an invariant which ensures that there are no child attributes. Tools should indicate this in an appropriate way, e.g. making the included template objects read-only, or allowing them to be edited while making it clear to users that the referenced template rather than a local copy is being edited.

5.2.4 Archetype Constraint Redefinition

Constraint redefinition is done in the same way as in a specialised archetype, with two exceptions:

- object node identifiers are not changed (in a specialised archetype, the code at0001 would be redefined to at0001.3 for example);
- 'cloning' of object nodes under container attributes can occur; cloned nodes have a template-wide unique *clone_id*.

Otherwise, the narrowing of archetype constraints, including occurrences, cardinality, existence, reference model type, and values is performed as for a specialised archetype.

5.2.4.1 Cloning

Cloned object nodes are represented by redefining an original `C_COMPLEX_OBJECT` object in an archetype with multiple instances of `C_COMPLEX_OBJECT`, each with the *clone_id* attribute set. In this situation, the *is_clone* flag of each such object will return `True`.

The semantics of cloning are similiar to specialisation into multiple nodes, i.e. a finer set of distinct sub-categories is being specified. Since this is being done in a template, the `C_OBJECT` *node_id* attribute is no longer available for further change, and the distinctions are indicated in the `C_DEFINED_OBJECT` *clone_id* attribute instead. Cloning is illustrated in FIGURE 7.

5.2.4.2 Terminology Subset Redefinition

The use of more constrained terminology subsets can be achieved in a template in the same way as within a specialised archetype, with the exception that instead of redefining the relevant constraint

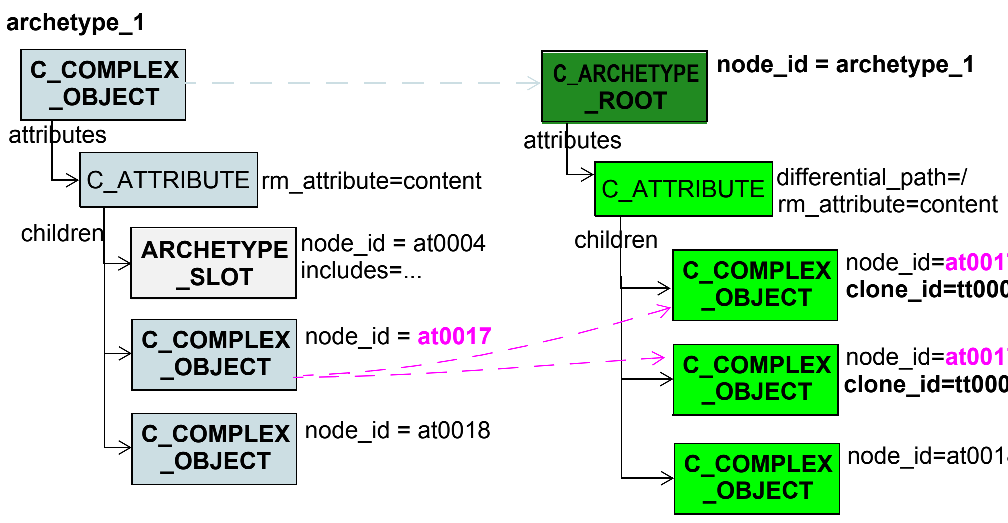


FIGURE 7 Template node cloning

code from say ac0001 to ac0001.1, it will be redefined instead to a template code such as tc0001, which is defined in the template ontology section.

5.2.5 Default Values

Default values are defined on the `C_DEFINED_OBJECT` in the AOM, and instances can be defined within a template on any descendant of this class. In practice, the use of default values in templates is normally limited to primitive types and other near-leaf complex types, including descendants of the `C_DOMAIN_TYPE` class. Default values are always of the reference model type constrained by the archetype object node to which they are attached.

In a template, the default value may be the only override set on a node. In terms of object structure, this will result in an empty `C_COMPLEX_OBJECT` or `C_PRIMITIVE_OBJECT`, to which the default value object is attached.

TBD_9: more here.

5.2.6 Conditional Rules

Template conditions are expressed in the form of logic statements, using the `RULE_STATEMENT` classes defined in the Archetype Object Model, and are attached to the template in the same way as for archetypes, i.e. via the *rules* attribute. The meaning of any such condition is that if present, and evaluated to True at runtime, the constraint to which it is attached will be considered to apply. Absence of a condition means that the constraint will always apply.

5.3 Template Validity

TBD_10: Template-wide validity: no changed node_ids

TBD_11:

5.4 Class Definitions

5.4.1 TEMPLATE Class

CLASS	**TEMPLATE**	
Purpose	The root object of an *open*EHR template. A `TEMPLATE` object carries at least one identifier, meta-data properties inherited from `DIFFERENTIAL_ARCHETYPE` and the *definition*, in a differential form very similar to that of a specialised archetype.	
Use	A `TEMPLATE` object can only be used to represent a template definition, not an operational termplate, which is the flat or expanded form.	
Inherit	`DIFFERENTIAL_ARCHETYPE`	
Attributes	**Signature**	**Meaning**
1 (redefined)	**definition**: `C_ARCHETYPE_ROOT`	Definition part of the template, containing constraints and other settings with respect to a set of archetypes.
Invariant		

5.4.2 C_TEMPLATE_ROOT Class

CLASS	**C_TEMPLATE_ROOT**
Purpose	The root object of a template reference within a slot. Referenced templates cannot be further constrained, so no child attributes are allowed.
Inherit	`C_ARCHETYPE_ROOT`
Invariant	**Attributes_validity**: attributes.is_empty

6 Examples

6.1 Constrain *open*EHR name attribute to Text

6.2 Template adds terminology subset to DV_TEXT

Original ADL:

DV_TEXT matches {}

want to add coding constraint like

DV_CODED_TEXT matches {[ac0003]} -- but do we want to require acnnnn + binding approach

plus may want to allow default to coding with any code, i.e.

DV_CODED_TEXT matches {[snomed::]}

6.3 No coding allowed on DV_TEXT constraint

how to remove the ability to allow a DV_CODED_TEXT on a DV_TEXT constraint in the archetype - just detect in the application, since there will be no details of subset or terminology set;

TBD_12: Q how to disinquish from the case where free coding allowed?

6.4 Override EVENT with INTERNAL_EVENT

Override with INTERNAL_EVENT that has width set and limited set of codes on math_function

6.5 Condition on Age or Sex

To Be Continued:

7 Serialisation

7.1 dADL

To Be Continued:

7.2 XML

To Be Continued:

8 The operational_template Package

8.1 Overview

The `operational_template` package is the fourth package within the archetype model, and is illustrated in context in FIGURE 8.

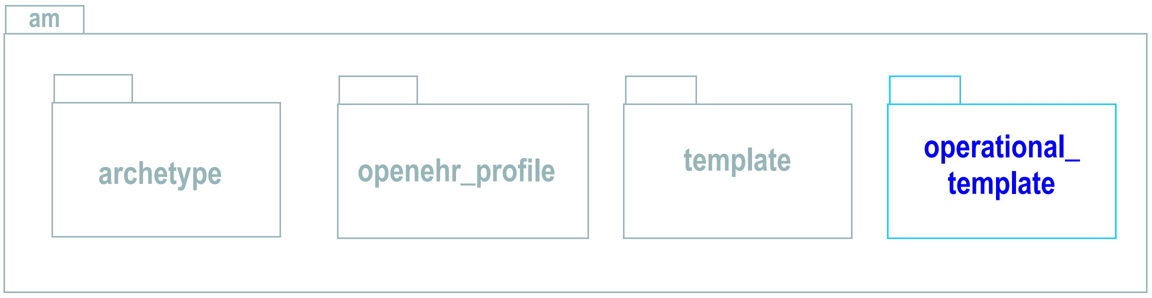


FIGURE 8 openehr.am.template Package in context

FIGURE 9 illustrates the `operational_template` package.

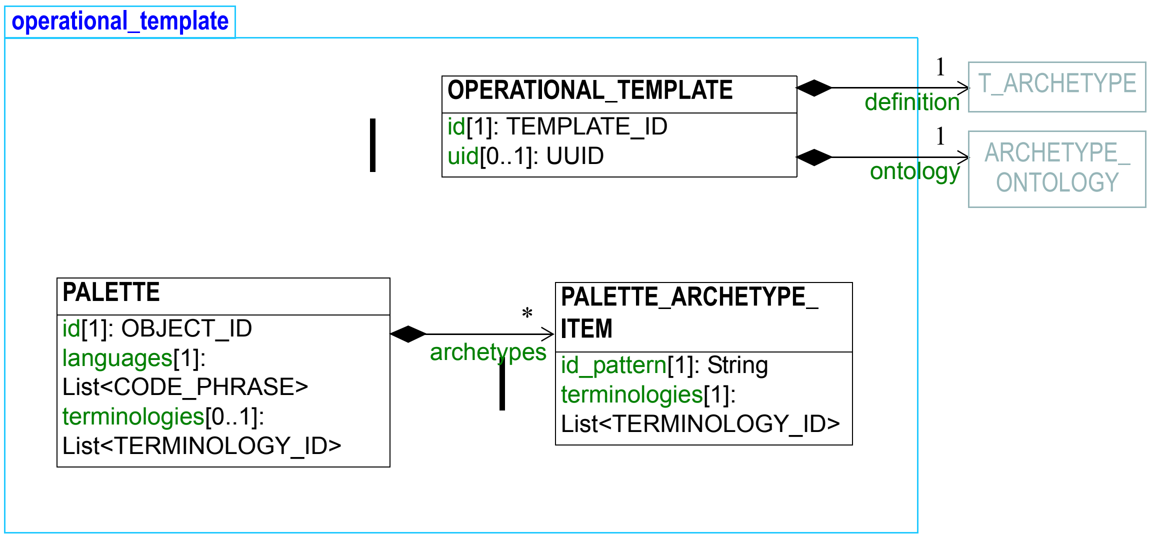


FIGURE 9 The openehr.am.template.operational_template Package

As can be seen from the model, an operational template has nearly the same form as a template definition. The following aspects are different:

- it is fully populated rather than being a differential artefact;
- it includes an ontology, which carries the merged ontologies from all archetypes used by the template;
- all internal references have been replaced by copies of the subtree to which they refer;
- the only `CONSTRAINT_REF` objects that remain are those that correspond to terminology subsets that will actually be obtained at runtime from a terminology server;
- template annotations are removed.

The operational template can be thought of as the flat-form version of a template definition, which is a differential artefact. The `operational_template` package also includes the *open*EHR palette, which defines the languages and terminologies to be used for a particular purpose.

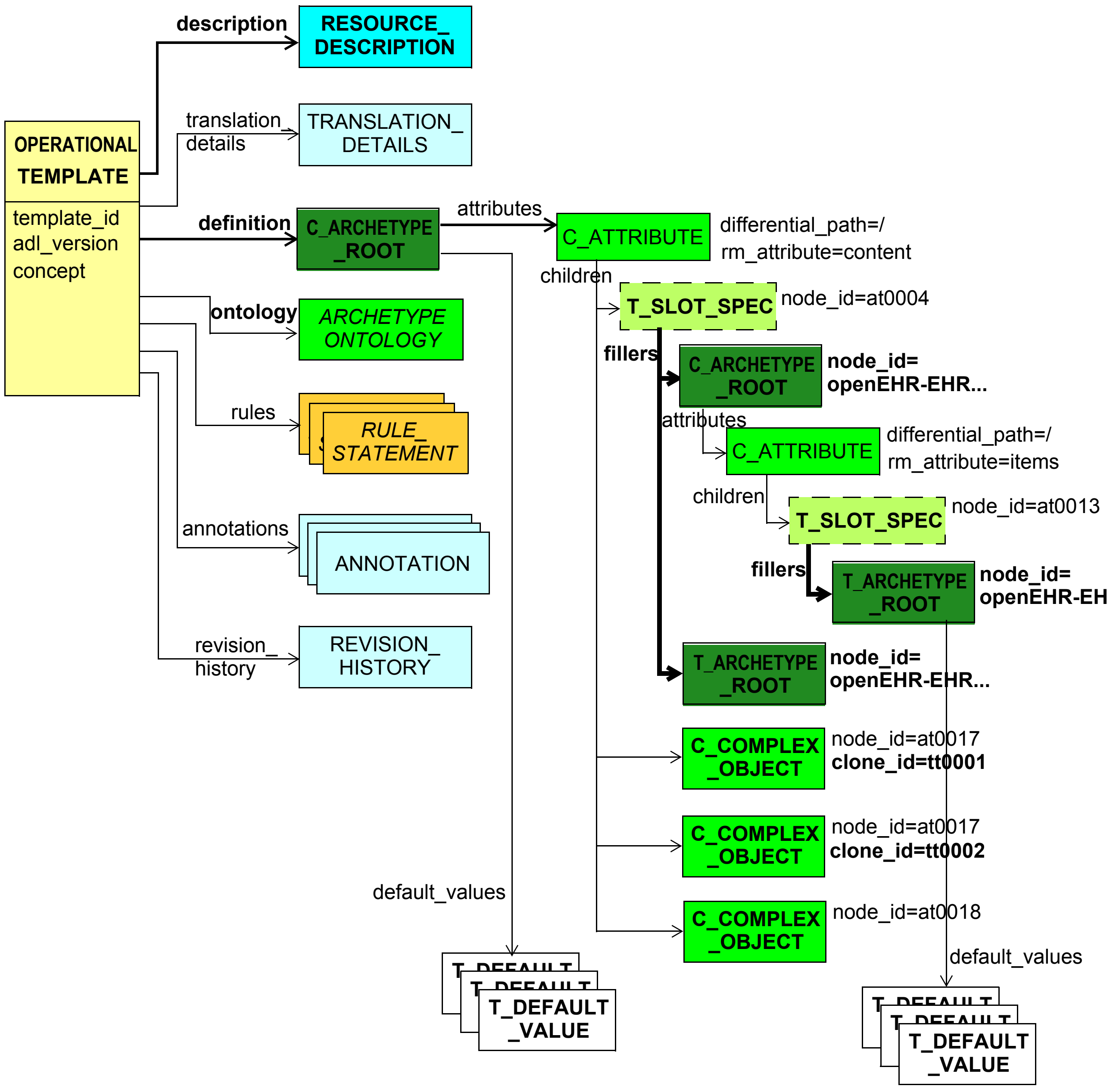


FIGURE 10 Typical Operational Template Object Structure

8.2 Design

Notes - in an operational template:

- use_nodes are expanded out to copies of the referenced nodes
- ac_code nodes remain as is, unless there is a terminology binding where we use a subtype of constraint_ref that has a terminology uri.

- unneeded languages are stripped
- unfilled slots: an ARCHETYPE_SLOT object is retained ?
- filled slots - how to leave slot open?
- ARCHETYPE_EXTERNAL_REF replaced by C_ARCHETYPE_ROOT
- ontology includes:
    - template codes section
    - section for each archetype, indexed by archetype id

8.3 Class Definitions

8.3.1 OPERATIONAL_TEMPLATE Class

CLASS	OPERATIONAL_TEMPLATE	
Purpose		
Use		
Abstract	Signature	Meaning
1	**id**: `TEMPLATE_ID`	
0..1	**uid**: `UUID`	
1	**definition**: `T_ARCHETYPE`	
Invariant	***id_valid***: id /= Void ***definition_valid***: definition /= Void	

8.3.2 PALETTE Class

CLASS	PALETTE	
Purpose		
Use		
Abstract	Signature	Meaning
1	**id**: `OBJECT_ID`	
1	**languages**: `List<String>`	
0..1	**terminologies**: `List<TERMINOLOGY_ID>`	

CLASS	PALETTE	
0..1	**archetypes**: `List <PALETTE_ARCHETYPE_ITEM>`	
Invariant	***id_valid***: palette_id /= Void ***languages_valid***: languages /= Void **and then not** languages.is_empty	

8.3.3 PALETTE_ARCHETYPE_ITEM Class

CLASS	PALETTE_ARCHETYPE_ITEM	
Purpose		
Use		
Abstract	Signature	Meaning
1	**id_pattern**: `String`	
1	**terminologies**: `List<TERMINOLOGY_ID>`	
Invariant	***id_pattern_valid***: id_pattern /= Void **and then not** id_pattern.is_empty ***terminologies_valid***: terminologies /= Void **and then not** terminologies.is_empty	

A References

Publications

1 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. OOPSLA 2002 workshop on behavioural semantics.
Available at http://www.deepthought.com.au/it/archetypes.html.

2 Beale T. *Archetypes: Constraint-based Domain Models for Future-proof Information Systems*. 2000. Available at http://www.deepthought.com.au/it/archetypes.html.

3 Beale T, Heard S. The Archetype Definition Language (ADL). See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/language/ADL/REV_HIST.html.

4 Heard S, Beale T. Archetype Definitions and Principles. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/principles/REV_HIST.html.

5 Heard S, Beale T. *The openEHR Archetype System*. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/system/REV_HIST.html.

Resources

6 *open*EHR. EHR Reference Model. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/top.html.

END OF DOCUMENT