*open*EHR
Release 1.0.2

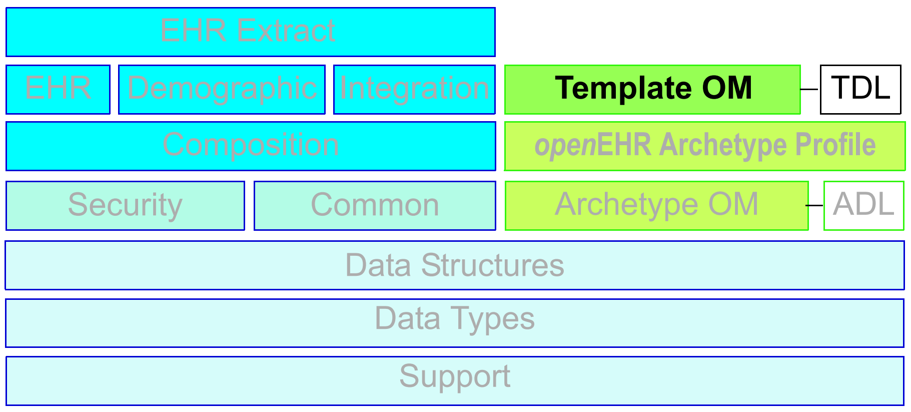
The *open*EHR Archetype Model

*open*EHR Templates

Editors: {T Beale, S Heard}[a]		
Revision: 1.0	*Pages:* 39	*Date of issue:* 20 Apr 2009
Status: TRIAL		

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 7
1.1 Purpose 7
1.2 Related Documents 7
1.3 Nomenclature 7
1.4 Status 8
1.5 Tools 8

2 Overview 9
2.1 Context 9
2.2 Computational Environment 9
2.3 What is a Template? 10
2.4 Template Definition 11
2.5 The Role of TDL 11

3 Requirements 12
3.1 Overview 12
3.2 Identification 12
3.3 Documentary Requirements 12
3.3.1 Descriptive Meta-data 12
3.3.2 Author Annotations 13
3.3.3 Institutional Meta-data and Links 13
3.4 User Interface Requirements 13
3.5 Semantic Requirements 13
3.5.1 Relationship to Archetypes 13
3.5.2 Slot Filling 13
3.5.3 Structural Constraints 14
3.5.4 Type Constraints 14
3.5.5 Leaf Value Constraints 14
3.5.6 Default Values 15
3.5.7 Conditional Constraints 15
3.5.8 Terminology Subsets 15
3.5.9 Pre-population of Terminology Subsets 15
3.6 Tooling Requirements 16

4 Template Definition Language (TDL) 17
4.1 Introduction 17
4.2 Basics 17
4.2.1 Reference Model Assumptions 17
4.2.2 Relationship to Archetypes 17
4.2.3 Node Identification 18
4.3 Header Sections 18
4.3.1 Template Identification Section 18
4.3.2 Language Section and Language Translation 18
4.4 Description 18
4.5 Definition 19
4.5.1 Overview 19
4.5.2 Slot Filling 19
4.5.3 Template Constraints 20

4.5.4 Object Renaming 21
4.5.5 Object Cloning 22
4.5.6 Default Values 23
4.5.7 Conditions 23
4.5.8 Terminology Binding 24
4.6 Annotations 24

5 The template Package 25
5.1 Overview 25
5.2 The Template Object 26
5.2.1 Template Identification 26
5.2.2 Meta-data 27
5.2.2.1 Template Description 27
5.2.2.2 Annotations 27
5.2.3 Validity 27
5.3 Template Definition 27
5.3.1 Overview 27
5.3.2 Slot-filling 28
5.3.4 Cloning 28
5.3.5 Terminology Subsets 28
5.3.6 Default Values 28
5.3.6.1 Primitive Types 28
5.3.6.2 Complex Types (limited) 28
5.3.7 Conditional Rules 29
5.4 Template Paths 29
5.5 Class Definitions 29

6 Examples 33
6.1 Constrain openEHR name attribute to Text 33
6.2 Template adds terminology subset to DV_TEXT 33
6.3 No coding allowed on DV_TEXT constraint 33
6.4 Override EVENT with INTERNAL_EVENT 33
6.5 Condition on Age or Sex 33

7 Serialisation 34
7.1 dADL 34
7.2 XML 34

8 The operational_template Package 35
8.1 Overview 35
8.2 Class Definitions 36
References 38

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 the third layer above archetypes and the reference model in the *open*EHR semantic 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.

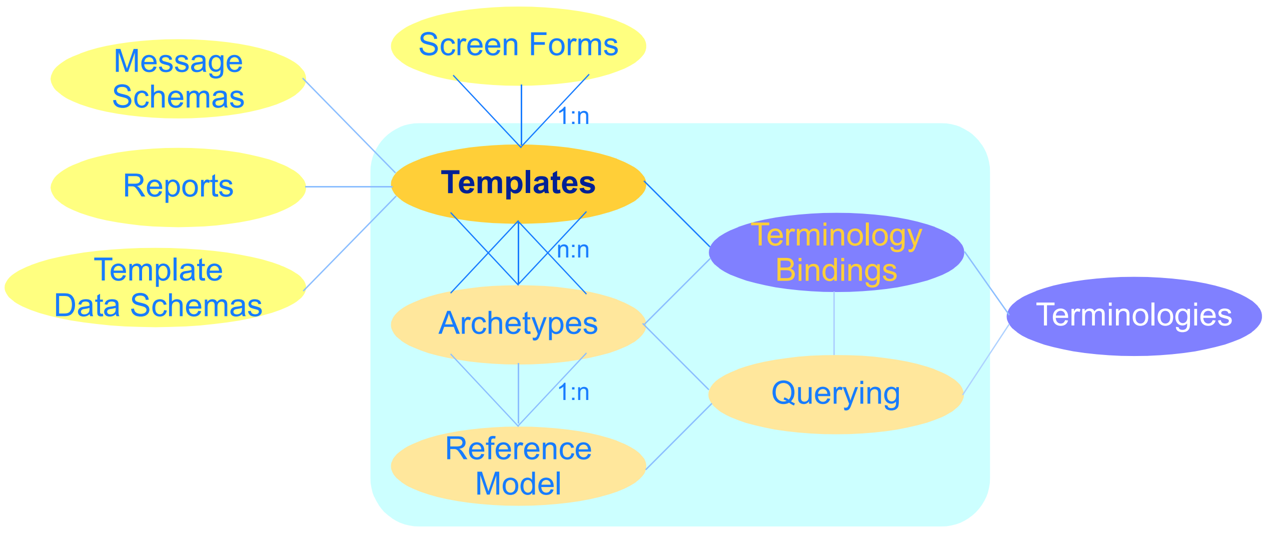


FIGURE 1 The *open*EHR Semantic Architecture

Templates are defined by the following specifications:

- the Template Definition Language (TDL);
- the Template Object Model (TOM);
- the Operational Template Model (OTM).

A related artefact is the *open*EHR *palette*, which which defines the local language and terminology preferences on a per-archetype basis. This is specified as part of the OTM. All of the above are defined in this document.

2.2 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.

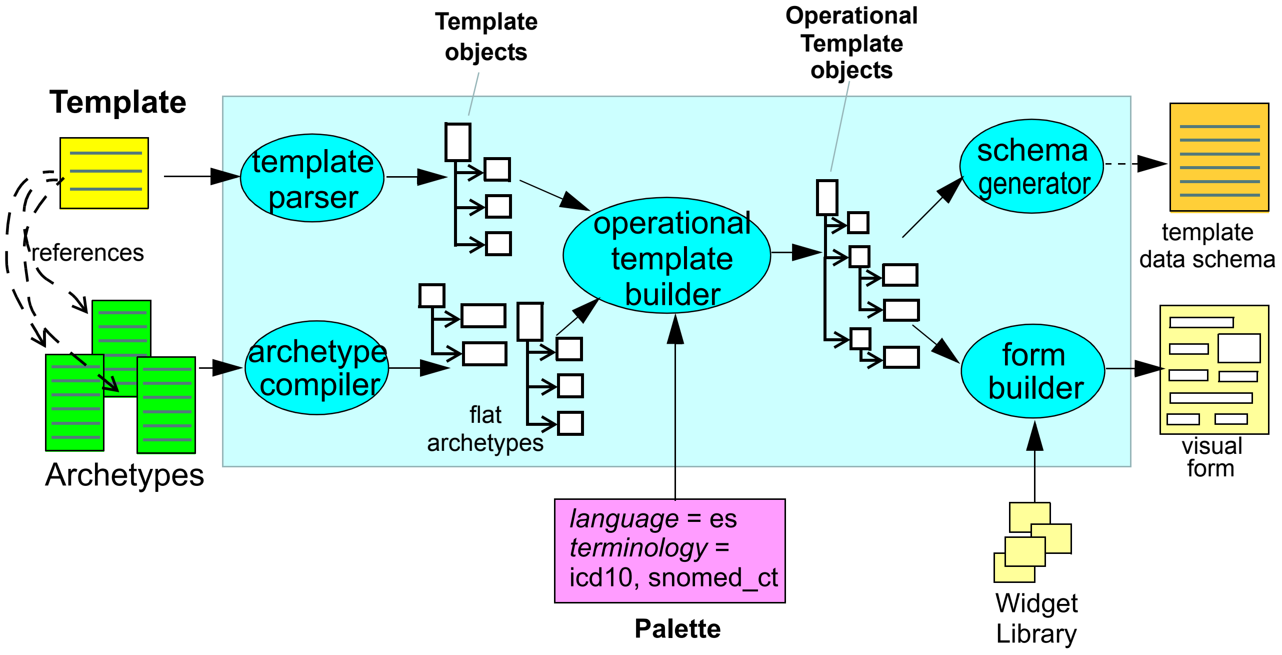


FIGURE 2 Template Tool Chain

2.3 What is a Template?

An *open*EHR template is an artefact that enables the content defined in archetypes to be used for a particular business purpose. 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 principal means of using archetypes in runtime systems.

This job of a template is to:

- aggregate archetypes into larger structures by indicating which archetypes should fill the slots of higher-level archetypes;
- choose elements (‘data points’) from archetypes for use in a template.

A template may aggregate a number of archetypes, e.g. 5 or more, but choose very few data points from each, thus having the effect of defining a small data set from a possibly very large number of data points defined in the original archetypes.

Templates embody the following semantics:

archetype slot-filling: choice of archetypes to make up a larger structure, specified via indicating identifiers of archetypes to fill slots in higher-level archetypes;

tightened constraints: tightening of other constraints, including cardinality, value ranges, terminology value sets and so on;

default values: choice of default values for use in templated structure at runtime;

meta-data: including node-level annotations.

2.4 Template Definition

In the technical sense, an *open*EHR template is a 'differential' artefact - i.e. a constraint model expressed with respect to other models. This has the practical effect that its contents are mainly references to archetypes, with associated constraint redefinitions and default values. The operational template 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 the 'meaning' of nodes, which is encoded in object node identifiers (at-codes).
- However, object nodes can be *renamed* and within container attributes, *cloned* with renaming.

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*.

2.5 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.

3 Requirements

3.1 Overview

Templates need to be able to perform three main functions. The first is to combine archetypes into larger structures according to the constraints expressed in archetype slots.

The second is to further constrain general-purpose archetypes for local use. In many cases, this involves 'removing' optional values and indeed whole parts of structures specified in an archetype or the underlying reference model (where not constrained by the archetypes being used). It also requires constraints on terminology use to be supported.

The third requirement is to be able 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.

TBD_1: the alternative requirement would be that author annotations could be made about any node in the resulting structure. This would be possible but would require a different solution.

TBD_2: annotation content could alternatively be something stronger like an openEHR DV_PARSEABLE which would allow syntax annotations like XML, HTML etc, but is likely to make annotations less interoperable.

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_3: template to 'form' relationship

TBD_4: 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 Structural Constraints

The following kinds of overrides on the structures expressed in archetypes need to be supported in templates:

- narrow the *existence* of a single-valued attribute from optional (0..1) within an archetype to either absent (0..0) or mandatory (1..1);
- narrow the *cardinality* of a multi-valued attribute, i.e. reduce the range of allowed items in a container, for example from 0..* defined in an archetype to 1..5;
- narrow the *occurrences* of an object, i.e. reduce the number of instances matching a particular object node within an archetype, e.g. from 0..10 to 0..1 or 0..1 to 0.

In each case, the template override range must be within the range defined by the archetype. These overrides provide the main means of removing unwanted elements from archetypes (i.e. converting optional nodes to 0..0) and for making other nodes mandatory (converting optional nodes to 1..1).

3.5.4 Type Constraints

Where the types of object nodes have been stated and possibly partially constrained in an archetype, a template can add a narrower version of the constraint in the form of a subtype. For example, if the the *open*EHR reference model type `EVENT` was specified in an archetype, a template could constrain it to `INTERVAL_EVENT`, and optionally constraint the values of certain attributes. For a single-valued attribute, the new constraint may be defined as:

- an override of an existing node, i.e. data at that node in an instance structure must conform only to the new constraint;
- an additional node, meaning that the data may conform either to the original constraint or to the new one.

In both cases, the new constraint must define a set of instances fully contained by the set defined by the original constraint.

3.5.5 Leaf Value Constraints

Leaf value constraints defined in archetypes are in the form of ranges, lists of values, and in the case of strings, patterns. All such constraints are equivalent to sets of possible values. Templates need to be able to refine value constraints that are defined in archetypes or the reference model in the following ways:

- by *redefining* an existing value constraint with an a new one of the same form, but matching a narrower set of values;
- by *excluding* certain values from a value constraint.

3.5.6 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`).

3.5.7 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 other 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.

3.5.8 Terminology Subsets

Within a constraint on a coded term (`CODE_PHRASE`), archetypes may contain references to external terminology resources, typically subsets. These are expressed in ADL by the use of ac-codes in the main definition, and bindings from the ac-codes to URIs referring to the resource, in the ontology section of an archetype. In the AOM, the ac-code node is expressed using a `CONSTRAINT_REF` object.. Terminology subsets need to be overridable in a template by a narrower terminology subset using the same method.

3.5.9 Pre-population of Terminology Subsets

In some cases it is desirable to be able to prepopulate 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_5: `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.

3.6 Tooling Requirements

TBD_6: `Nested templates in a different colour`

TBD_7: `cut and paste of a template, template section`

4 Template Definition Language (TDL)

4.1 Introduction

An 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 as follows:

```
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 Reference Model Assumptions

*open*EHR Templates, for reasons of pratical utility make some assumptions about the underlying reference model on which they are based which archetypes do not need to make, but which are satisfied by the *open*EHR reference model. Firstly, they assume the presence of the *name* attribute on those nodes for which it is defined in the *open*EHR RM, i.e. any descendant of the `LOCATABLE` class. This includes most classes in the reference model. This assumption is made in order that templates can specifically set or redefine the name value on archetype nodes used within the template, enabling the definition of multiple *clones* of a given archetype node, each clone carrying a distinct name. Renaming and cloning is described in section 4.5.4 and section 4.5.5.

The second assumption from the reference model is the existence of the abstract type `DATA_VALUE` and descendants, as used in *open*EHR. This assumption limits default values in templates to being expressed as either `DATA_VALUE` or primitive type instances, which satisfies the needs of *open*EHR.

TBD_8: this assumption is invalid if there are types that are not DATA_VALUEs or primitive types that need to be constrained from RM classes in archetypes.

4.2.2 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.

4.2.3 Node Identification

A template cannot redefine object node identifiers found within archetypes it uses. As a result, it has no ontology section as found in an archetype, since it defines no terms of its own. 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.v1.2.0
    e65542c8-881e-48c4-9f4a-75deaeaf3be8
```

TBD_9: the form for template identifiers may take some time to be agreed, but in fact has no practical effect on the rest of the semantics or model described here - the model simply treats them as strings.

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 containing the following elements:

- authoring organisation reverse domain name;
- identifier of reference model class being templated;
- logical name of the template;
- version identifier in the form 'vn.m.p' where n.m.p is a three-part numerical version 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.

TBD_10: when to use each type of id

TBD_11: how does versioning work in template ids?

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 ∈ {-- 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.

```
    SECTION[openEHR-EHR-SECTION.history_medical_surgical.v1] ∈ {
        /items[at0001] ∈ {
            use_archetype EVALUATION[openEHR-EHR-EVALUATION.problem.v1]
            use_template EVALUATION[uk.nhs.cfh::openEHR-EHR-EVALUATION.ed-
                                                            diagnosis.v1]
            use_archetype EVALUATION[openEHR-EHR-EVALUATION.clin_synopsis.v1]
        }
    }
```

TBD_12: how to indicate closed or open status?

Slots can be recursively filled in this way, 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:

```
    COMPOSITION[openEHR-EHR-COMPOSITION.xxx.v1] ∈ {
        /content ∈ {
            use_archetype SECTION[openEHR-EHR-SECTION.yyy.v1] ∈ {
                /items[at0001] ∈ {
                    use_template EVALUATION[uk.nhs.cfh::
                                        openEHR-EHR-EVALUATION.xx.v1]
                    use_archetype EVALUATION[openEHR-EHR-EVALUATION.xx.v1]
                }
                /items[at0002] ∈ {
                    use_archetype EVALUATION[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.

```
     SECTION[openEHR-EHR-SECTION.history_medical_surgical.v1] ∈ {
         /items[at0001] ∈ {
            use_archetype EVALUATION[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[openEHR-EHR-EVALUATION.clin_synopsis.v1] ∈
            {
               /data/items ∈ {
                  ELEMENT[at0003] ∈ {
                     ...
                  }
               }
            }
         }
     }
To Be Continued:
```

4.5.4 Object Renaming

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_13: 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. The template package defines the definition form of an *open*EHR template.

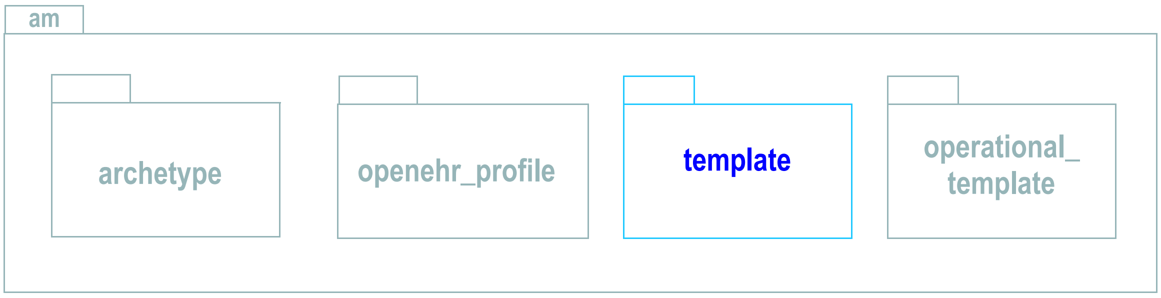


FIGURE 3 openehr.am.template Package in context

The template package defines the object model of a template and is shown in FIGURE 4.

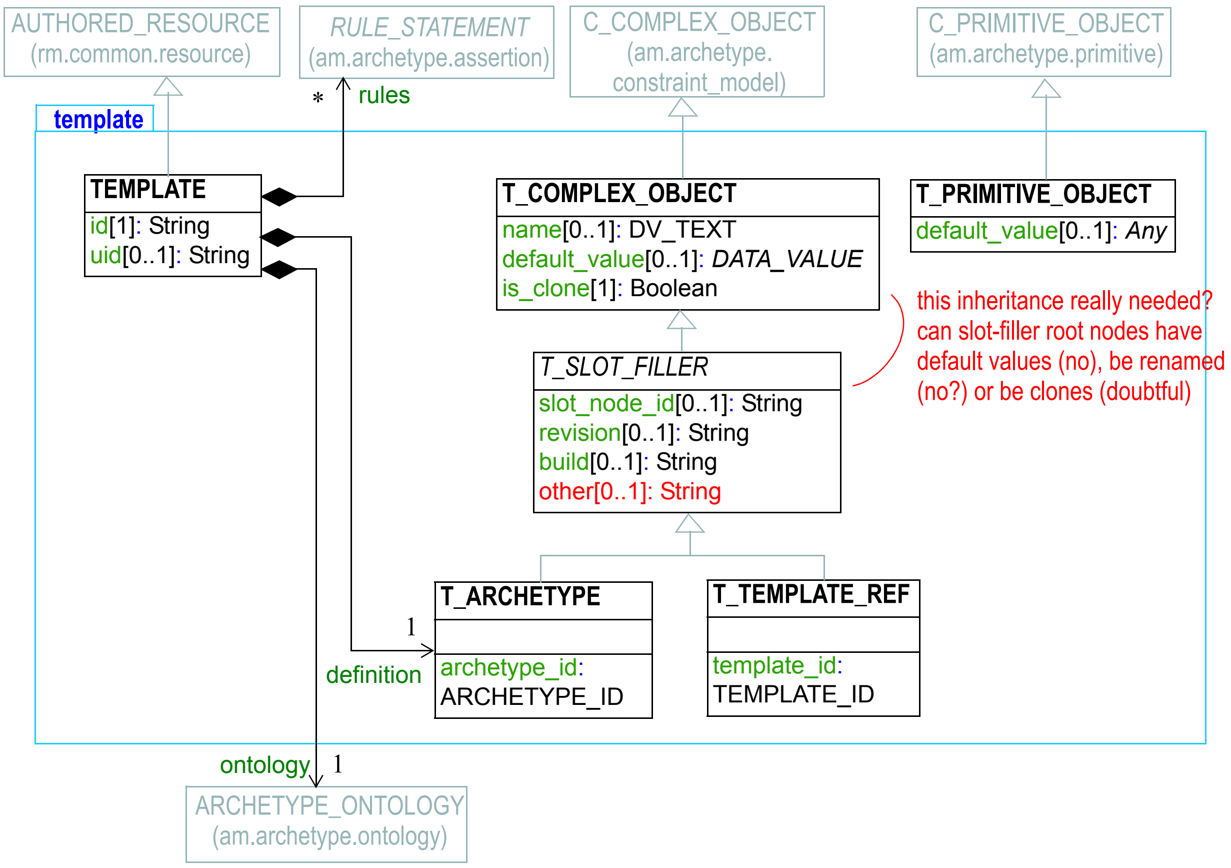

AUTHORED_RESOURCE (rm.common.resource)
RULE_STATEMENT (am.archetype.assertion)
C_COMPLEX_OBJECT (am.archetype. constraint_model)
C_PRIMITIVE_OBJECT (am.archetype.primitive)
template
* rules
TEMPLATE
id[1]: String
uid[0..1]: String
T_COMPLEX_OBJECT
name[0..1]: DV_TEXT
default_value[0..1]: DATA_VALUE
is_clone[1]: Boolean
T_PRIMITIVE_OBJECT
default_value[0..1]: Any
this inheritance really needed? can slot-filler root nodes have default values (no), be renamed (no?) or be clones (doubtful)
T_SLOT_FILLER
slot_node_id[0..1]: String
revision[0..1]: String
build[0..1]: String
other[0..1]: String
T_ARCHETYPE
archetype_id: ARCHETYPE_ID
T_TEMPLATE_REF
template_id: TEMPLATE_ID
1 definition
ontology 1
ARCHETYPE_ONTOLOGY (am.archetype.ontology)

FIGURE 4 openehr.am.template package

The model has been defined as a small number of classes in addition to the *open*EHR Archetype Object Model (AOM), enabling the existing semantics of the AOM to be reused.

FIGURE 5 illustrates a typical archetype object structure. Mandatory parts are shown with a bold association.

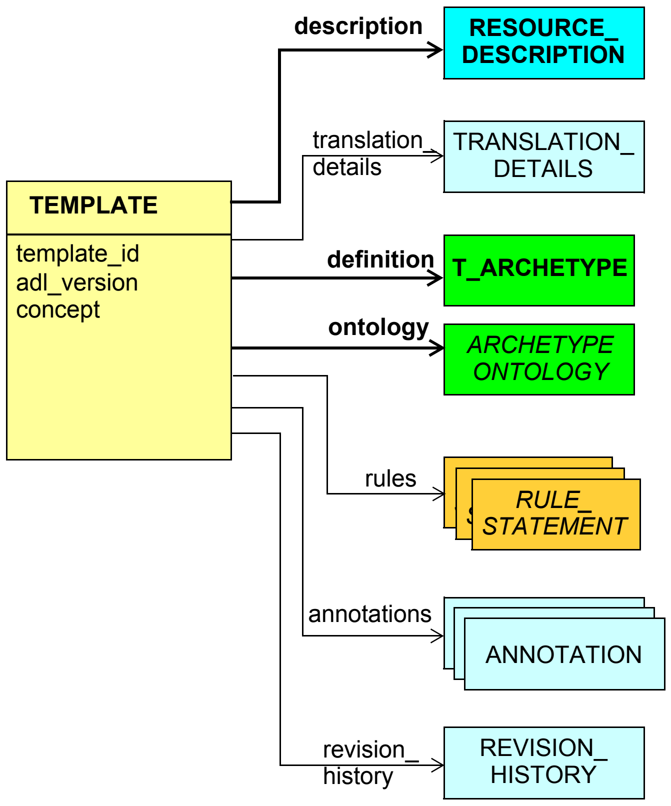


FIGURE 5 Template Object Structure

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

5.2 The Template Object

From a structural point of view, a template consists of identifiers, meta-data (inherited from the AUTHORED_RESOURCE class), a *definition* and optionally a list of *annotations*. As with an archetype, the *definition* carries the main declarative content of a template, and is defined as a T_ARCHETYPE, a descendant of C_COMPLEX_OBJECT.

5.2.1 Template Identification

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, in the rm.support.identification package.

5.2.2 Meta-data

5.2.2.1 Template Description

Templates support two types of 'built-in' meta-data, both due to inheritance from the AUTHORED_RESOURCE class (rm.common.resource). The first is the same meta-data as is found in the description section of an archetype, and includes purpose, use, misuse, references to original resources and so on. The capability to translate these, and indeed to define the original language and translations in general of a template come from this class.

5.2.2.2 Annotations

Another meta-data facility is available at the node level of templates. This takes the form of a list of *annotations*, also inherited from the AUTHORED_RESOURCE class. Each annotation is associated with a template node via the use of the ANNOTATION.*path* attribute, which must carry a full template path (see section 5.4 on page 29). The ANNOTATION.*items* attribute defines the annotions as a keyed list of Strings.

Annotations are defined as a separate list to allow 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 / release. 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.

To Be Continued: example

5.2.3 Validity

TBD_14: Template-wide validity: no changed node_ids

5.3 Template Definition

5.3.1 Overview

The main content of a template is defined as a hierarchy of C_OBJECT and C_ATTRIBUTE objects, as for specialised archetypes. All redefinitions possible in a specialised archetype are available in a template, with the exception that object node identifiers are not changed. To handle template-only semantics, the template package adds the specialisations of the C_COMPLEX_OBJECT class from the AOM, namely T_COMPLEX_OBJECT, T_SLOT_FILLER, T_ARCHETYPE and T_TEMPLATE_REF and a specialisation of C_PRIMITIVE_OBJECT, T_PRIMITIVE_OBJECT. The first of these allows a C_COMPLEX_OBJECT node within a template to carry a name, a default value and to be marked as a clone.

The T_ARCHETYPE specialisation is used to represent the root node of an archetype within a template, while the T_TEMPLATE_REF type is used to represent a reference to an existing template within a template, supporting reuse of templates by other templates. Both of these types inherit from T_SLOT_FILLER, which provides identifying details for the archetype or template filling the slot. Both types use the inherited C_OBJECT attribute *node_id* to carry an archetype identifier and a template identifier respectively, rather than the at-codes used on normal nodes within archetypes. Functions are provided to return these identifiers.

The T_PRIMITIVE_OBJECT class supports the addition of default values to primitive object constraints in templates.

5.3.2 Slot-filling

A template definition may completely specify the archetypes filling a slot, specify only some, or make no statements about a slot, leaving it open until a close-to-runtime generation of an operational template. An archetype slot is filled in a template using one or more instances of the classes T_ARCHETYPE and T_TEMPLATE_REF. The multiplicity is governed by the *occurrences* range on the ARCHETYPE_SLOT object from the original archetype. As for any other C_OBJECT, an ARCHETYPE_SLOT generally carries a *node_id*, if it is defined in a position in the archetype requiring one, and this is almost always the case because slots tend to be defined under multiply-valued attributes. In such cases, each slot in the parent archetype under a given attribute carries its own at-code, and as a consequence, each slot filler object must use this at-code to indicate which slot it is filling. The T_SLOT_FILLER *slot_node_id* attribute provides this facility.

TBD_15: inherited 'closed' flag required from AOM to mark container nodes as open or closed for further additions at runtime? Does this idea even make sense?

Each filler can be either an archetype, or an included template. Templates are included by reference, and their content is not further modifiable by the referencing template. 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.

5.3.3 Renaming

A renamed object node in a template is represented by an instance of T_COMPLEX_OBJECT with the *name* attribute set as required to either a DV_TEXT or DV_CODED_TEXT instance.

5.3.4 Cloning

Cloned object nodes are represented using instances of T_COMPLEX_OBJECT with the *name* attribute set as for renaming, and the *is_clone* flag set to True.

5.3.5 Terminology Subsets

To Be Continued: , i.e. by replacing the ac-code with another that is bound to a subset that is known to be subsumed by the original defined in the archetype. The subsumption relationship is asserted in the terminology subset service, and cannot be directly validated in a template.

5.3.6 Default Values

5.3.6.1 Primitive Types

A default value of a primitive type can be associated with a primitive object constraint node in an archetype via the use of T_PRIMITIVE_OBJECT, which is a template variant of C_PRIMITIVE_OBJECT used specifically for the purpose. The type of the *default_value* attribute conforms to the type specified in the inherited attribute *rm_type_name*.

5.3.6.2 Complex Types (limited)

In the same fashion as for primitive types, a default value can be stated for complex object nodes whose reference model type conforms to the *open*EHR type DATA_VALUE. This is a more limited aspect of the template model, and is based on experience with the need for default values being limited to 'data type' instances, which are constrained in archetypes with C_COMPLEX_OBJECT or C_DOMAIN_TYPE object nodes.

5.3.7 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.

5.4 Template Paths

All constraints and default values defined in a template are associated with paths, indicating which node in an archetype they apply to. Since templates enable hierarchically composed structures of archetypes, expressed by the 'chaining' of archetypes chosen to fill slots, all paths referenced in templates are archetype paths, i.e. stated with respect to the archetype identifier of the archetype within which a set of template constraints falls.

In order to identify a template node from *outside* the template, a full *template path* must be used. This is constructed by concatenating the archetype paths together, each prepended by its archetype id. Paths are built the same way whether the archetypes are directly referenced in the template, or other templates are included by reference. Template paths may be used to refer to specific nodes or node groups in a templated data structure (i.e. within operational data), or may be used for design purposes.

A template path is not guaranteed to uniquely identify a node in runtime data if the name attributes are not all set within the template (i.e. if they were left open to be set at runtime), but it is guaranteed to uniquely identify a node within a template (or to be technically correct, within the operational template generated from a template definition).

5.5 Class Definitions

5.5.1 TEMPLATE Class

CLASS	**TEMPLATE**	
Purpose	The root object of an *open*EHR template definition. A TEMPLATE object carries at least one identifier, meta-data properties inherited from AUTHORED_RESOURCE 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	AUTHORED_RESOURCE	
Attributes	**Signature**	**Meaning**
1	**id**: TEMPLATE_ID	Human readable multi-axial structured identifier for the template.
0..1	**uid**: UUID	Optional Guid identifier for the template.

CLASS	**TEMPLATE**	
1	**definition**: `T_ARCHETYPE`	Definition part of the template, containing constraints and other settings with respect to a set of archetypes.
Invariant	***id_validity***: id /= Void ***definition_validity***: definition /= Void	

5.5.2 T_COMPLEX_OBJECT Class

CLASS	**T_COMPLEX_OBJECT**	
Purpose	A specialisation of `C_COMPLEX_OBJECT` that allows a template to associate a *name* and/or *default_value* with a complex object archetype node. A flag is also included to indicate whether the object is a clone of a complex object defined within a container.	
Inherit	`C_COMPLEX_OBJECT`	
Attributes	**Signature**	**Meaning**
0..1	**name**: `DV_TEXT`	Optional name override for corresponding archetype object constraint. The value is used to set the *name* attribute inherited from `LOCATABLE` in the corresponding reference model object at runtime.
0..1	**default_value**: *`DATA_VALUE`*	Optional default value for an object constraint. Limited to subtypes of the *open*EHR `DATA_VALUE` type. The type of the runtime object must match or conform to the type mentioned in the *rm_type_name* attribute inherited from `C_OBJECT`.
1	**is_clone**: `Boolean`	Flag to indicate if this object is a clone of a corresponding object constraint defined under a container attribute in an archetype. If False for a container object, this template object is a redefinition by renaming.
1	**hide_in_ui**: `Boolean`	Flag to indicate if this object should be hidden when rendering to the user interface. **EXAMPLE REQUIRED**.
Invariant	***basic_validity***: name /= Void **xor** default_value /= Void ***is_clone_validity***: is_clone **implies** name /= Void **and** parent.generating_type.is_equal("C_MULTIPLE_ATTRIBUTE") ***default_value_validity***: default_value.conforms_to_type(rm_type_name)	

5.5.3 T_ARCHETYPE Class

CLASS	T_ARCHETYPE	
Purpose	A variant of `T_COMPLEX_OBJECT` that represents an archetype root node at a slot position within a template. Instead of a normal at-code in the inherited *node_id* attribute, a `T_ARCHETYPE` carries an archetype identifier string.	
Use	In a template, a `T_ARCHETYPE` is only used within an archetype slot of another archetype higher in the compositional structure. The archetype identifier must match the slot pattern.	
Inherit	`T_COMPLEX_OBJECT`	
Functions	**Signature**	**Meaning**
1	**id**: `ARCHETYPE_ID`	Function returning the contents of the inherited node_id attribute as an `ARCHETYPE_ID`.
Invariant	***node_id_validity***: `{ARCHETYPE_ID}`.**valid_id**(node_id)	

5.5.4 T_TEMPLATE_REF Class

CLASS	T_TEMPLATE_REF	
Purpose	A variant of `T_COMPLEX_OBJECT` that represents another template within a template. Instead of a normal at-code in the inherited *node_id* attribute, a `T_TEMPLATE_REF` carries a template identifier string. No redefinitions can be made below a template reference, hence the attributes = Void invariant.	
Use	In a template, a `T_TEMPLATE_REF` is only used within an archetype slot of another archetype higher in the compositional structure. The root archetype identifier within the referenced template must match the slot pattern.	
Inherit	`T_COMPLEX_OBJECT`	
Functions	**Signature**	**Meaning**
1	**id**: `TEMPLATE_ID`	Function returning the contents of the inherited *node_id* attribute as an `TEMPLATE_ID`.
Invariant	***node_id_validity***: `{TEMPLATE_ID}`.valid_id(node_id) ***basic_validity***: attributes = Void	

5.5.5 T_PRIMITIVE_OBJECT Class

CLASS	T_PRI MITIVE_OBJECT	
Purpose	A template variant of `C_PRIMITIVE_OBJECT` that enables a default value of a primitive type to be defined for the corresponding archetype node.	
Inherit	`C_PRIMITIVE_OBJECT`	
Attributes	Signature	Meaning
1	**default_value**: *Any*	Default value defined for the corresponding archetype object.
Invariant	***default_value_validity***: default_value /= Void **and** default_value.conforms_to_type(rm_type_name)	

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_16: 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 6.

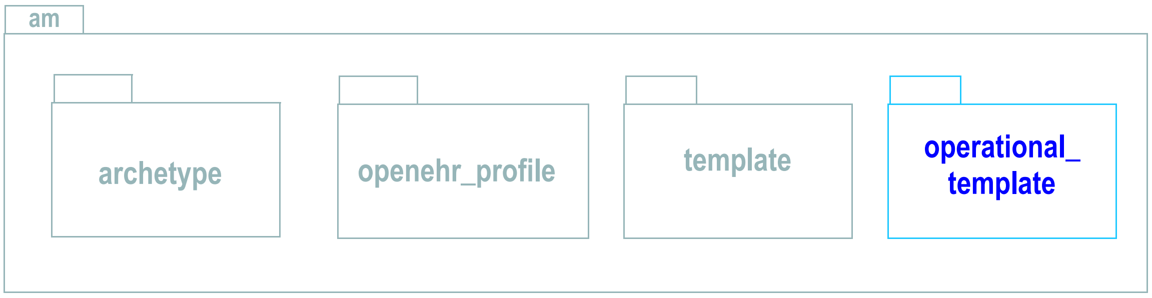


FIGURE 6 openehr.am.template Package in context

FIGURE 7 illustrates the `operational_template` package.

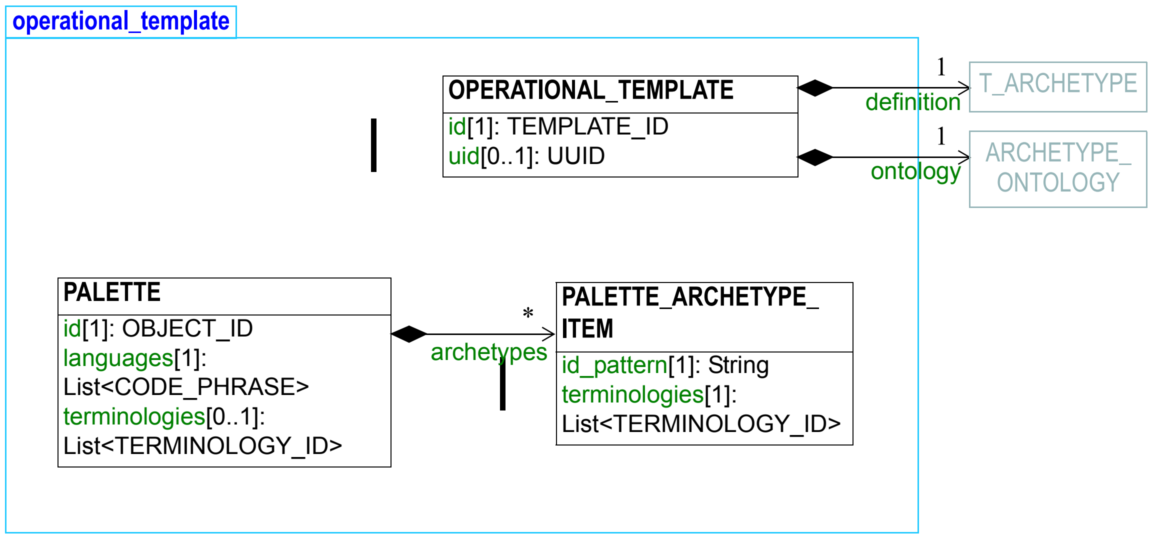


FIGURE 7 The openehr.am.template.operational_template Package

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.

8.2 Class Definitions

8.2.1 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	

8.2.2 PALETTE Class

CLASS	PALETTE	
Purpose		
Use		
Abstract	Signature	Meaning
1	**id**: `OBJECT_ID`	
1	**languages**: `List<String>`	
0..1	**terminologies**: `List<TERMINOLOGY_ID>`	
0..1	**archetypes**: `List <PALETTE_ARCHETYPE_ITEM>`	
Invariant	***id_valid***: palette_id /= Void ***languages_valid***: languages /= Void **and then not** languages.is_empty	

8.2.3 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