The *open*EHR Archetype Model

*open*EHR Archetype Profile

Editors:T Beale[1]

Revision: 1.0.0

Pages: 25

1. Ocean Informatics Australia

Copyright Notice

Amendment Record

Issue	Details	Who	Date
R E L E A S E 1.0.1			
1.0.0	**CR-000200**: Correct Release 1.0 typographical errors. Global changes to this document. Fix invariants in C_QUANTITY classes. Correct C_QUANTITY.*property* to CODE_PHRASE. Correct invariants for C_CODED_TEXT; correct inheritance for C_DV_ORDERED. Corrected C_QUANTITY_ITEM class. Corrected errors in DV_STATE model by adding 2 new classes.	T Beale, D Lloyd, R Chen	09 Nov 2006
	CR-000219: Use constants instead of literals to refer to terminology in RM.	R Chen	
	CR-000224: Relax semantics of C_QUANTITY etc to allow no constraint.	S Heard	
	CR-000226: Rename C_CODED_TEXT to C_CODE_PHRASE.	T Beale	
R E L E A S E 1.0			
R E L E A S E 0.95			
0.5	**CR-000127**. Restructure archetype specifications. Initial Writing.	T Beale	05 Feb 2005

Acknowledgements

The work reported in this paper has been funded by University College London and Ocean Informatics Pty Ltd, Australia.

Table of Contents

1 Introduction

1.1 Purpose

This document describes the *open*EHR Archetype Profile (AP), which defines custom constraint classes for use with the generic archetype object model (AOM). The intended audience includes:

- Standards bodies producing health informatics standards
- Software development organisations using *open*EHR
- Academic groups using *open*EHR
- The open source healthcare community
- Clinical and domain modelling specialists.

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Architecture Overview

Prerequisite documents for reading this document include:

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

1.3 Status

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

This document is available at http://svn.openehr.org/specification/TAGS/Release-1.0/publishing/architecture/am/openehr_archetype_profile.pdf.

The latest version of this document can be found at http://svn.openehr.org/specification/TRUNK/publishing/architecture/am/openehr_archetype_profile.pdf.

1.4 Peer review

Known omissions or questions are indicated in the text with a "to be determined" paragraph, as follows:

TBD_1: (example To Be Determined paragraph)

Areas where more analysis or explanation is required are indicated with "to be continued" paragraphs like the following:

To Be Continued: more work required

Reviewers are encouraged to comment on and/or advise on these paragraphs as well as the main content. Please send requests for information to info@openEHR.org. Feedback should preferably be provided on the mailing list openehr-technical@openehr.org, or by private email.

2 Overview

2.1 Background

An underpinning principle of *open*EHR is the use of archetypes and templates, which are formal models of domain content, and are used to control data structure and content during creation, modificatoin and querying. The elements of this architecture are twofold.

- The *open*EHR Reference Model (RM), defining the structure and semantics of information in terms of information models (IMs). The RM models correspond to the ISP RM/ODP information viewpoint, and define the data of *open*EHR EHR systems. The information model is designed to be invariant in the long term, to minimise the need for software and schema updates.
- The *open*EHR Archetype Model (AM), defining the structure and semantics of archetypes and templates. The AM consists of the archetype language definition language (ADL), the Archetype Object Model (AOM) and the *open*EHR Archetype profile (oAP).

The purpose of the ADL is to provide an abstract syntax for textually expressing archetypes and templates. The AOM defines the object model equivalent of ADL. It is completely *generic*, meaning that it can be used to express archetypes for any reference model in a standard way. ADL and the AOM are brought together in an ADL parser: a tool which can read ADL archetype texts, and whose parse-tree (resulting in-memory object representation) is instances of the AOM.

The purpose of the *open*EHR Archetype Profile, the subject of this document, is to define custom archetype classes and in some cases, custom syntax equivalents (essentially shorthands) that can be used instead of the AOM generic classes for archetyping certain RM classes.

2.2 Design Approach for Custom Types

A situation in which standard ADL falls short is when the required semantics of constraint are different from those available naturally from the standard approach. Consider a reference model type `QUANTITY`, shown at the top of FIGURE 1, which could be used to represent a person's age in an archetype.

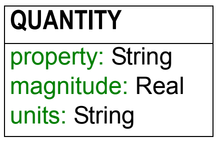

FIGURE 1 QUANTITY reference model type

A typical ADL constraint to enable `QUANTITY` to be used to represent age in clinical data can be expressed in natural language as follows:

```
property matches "time"
units matches "years" or "months"
if units is "years" then magnitude matches 0..200 (for adults)
if units is "months" then magnitude matches 3..36 (for infants)
```

The standard ADL expression for this requires the use of multiple alternatives, and would be as follows:

```
age matches {
        QUANTITY matches {
           property matches {"time"}
           units matches {"years"}
           magnitude matches {|0.0..200.0|}
        }
        QUANTITY matches {
           property matches {"time"}
           units matches {"months"}
           magnitude matches {|3.0..12.0|}
        }
    }
```

While this is a perfectly legal approach, it is not the most natural expression of the required constraint, since it repeats the constraint of *property* matching “time”. It also makes processing by software slightly more difficult than necessary.

A more powerful possibility is to introduce a new class into the archetype model, representing the concept “constraint on QUANTITY”, which we will call C_QUANTITY here. Such a class fits into the class model of archetypes (see the *open*EHR AOM), inheriting from the class C_DOMAIN_TYPE. The C_QUANTITY class is illustrated in FIGURE 2, and corresponds to the way constraints on QUANTITY objects are often expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

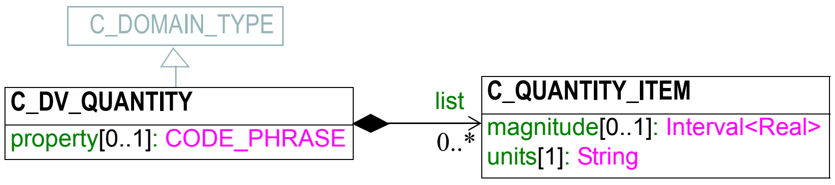

FIGURE 2 C_QUANTITY Type

The question now is how to express a constraint corresponding to this class in an ADL archetype. The solution is logical, and uses standard ADL. Consider that a particular constraint on a QUANTITY must be *an instance* of the C_QUANTITY type. An instance of any class can be expressed in ADL using the dADL sytnax (ADL’s serialised object syntax) at the appropriate point in the archetype, as follows:

```
    value matches {
        C_QUANTITY <
           property = <[openehr::128]>
           list = <
              items = <
                 [1] = <
                    units = <"yr">
                    magnitude = <|0.0..200.0|>
                 >
                 [2] = <
                    units = <"mth">
                    magnitude = <|1.0..36.0|>
                 >
              >
           >
        >
    }
```

This approach can be used for any custom type which represents a constraint on a reference model type. Since the syntax is generic, only one change is needed to an ADL parser to support C_XXX types in ADL texts. The syntax rules are as follows:

- the dADL section occurs inside the {} block where its standard ADL equivalent would have occurred (i.e. no other delimiters or special marks are needed);
- the dADL section must be 'typed', i.e. it must start with a type name, which should correspond directly to a reference model type;
- the dADL instance must obey the semantics of the custom type of which it is an instance, i.e. include the correct attribute names and relationships.

It should be understood of course, that just because a custom constraint type has been defined, it does not need to be used to express constraints on the reference model type it targets. Indeed, any mixture of standard ADL and dADL-expressed custom constraints may be used within the one archetype.

2.2.1 Custom Syntax

A dADL section is not the only possibility for expressing a custom constraint type. A useful alternative is a custom addition to the ADL syntax. Custom syntax can be smaller, more intuitive to read, and easier to parse than embedded dADL sections. A typical example of the use of custom syntax is to express constraints on the type CODE_PHRASE in the *open*EHR reference model (rm.data_types package). This type models the notion of a 'coded term', which is ubiquitous in clinical computing. The standard ADL for a constraint on a CODE_PHRASE is as follows:

```
defining_code matches {
    CODE_PHRASE matches {
        terminology_id matches {"local"}
        code_string matches {"at0039"} -- lying
    }
    CODE_PHRASE matches {
        terminology_id matches {"local"}
        code_string matches {"at0040"} -- sitting
    }
}
```

However, as with QUANTITY, the most typical constraint required on a CODE_PHRASE is factored differently from the standard ADL - the need is almost always to specify the terminology, and then a set of *code_strings*. A type C_CODE_PHRASE type can be defined as shown in FIGURE 3.

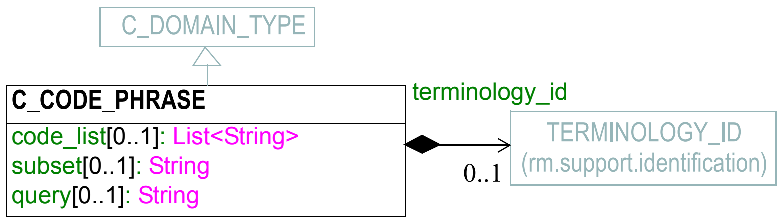

FIGURE 3 C_CODE_PHRASE

Using the dADL section method, including a C_CODE_PHRASE constraint would require the following section:

```
code matches {
    C_CODE_PHRASE <
        terminology_id = <
            value = <"local">
        >
```

```
            code_list = <
                ["1"] = <"at0039">
                ["2"] = <"at0040">
            >
        >
    }
```

Although this is perfectly legal, a more compact and readable rendition of this same constraint is provided by a custom syntax addition to ADL, which enables the above example to be written as follows:

```
    defining_code matches {
        [local::
            at0039,
            at0040]
    }
```

The above syntax should be understood as an extension to the ADL grammar, and an archetype tool supporting the extension needs to have a modified parser.

While these two ADL fragments express exactly the same constraint, the second is shorter and clearer.

2.3 Package Structure

The *open*EHR Archetype Profile model is defined in the package `am.openehr_profile`, illustrated in FIGURE 4. It is shown in the context of the *open*EHR am and `am.archetype` packages. The internal structure of the package mimics the structure of the reference model it profiles, i.e. the *open*EHR reference model. This is done to make software development easier, even though the package structure may be sparsely populated. Packages need only be defined where there are custom types to be defined; the only ones currently defined are in the `data_types` package.

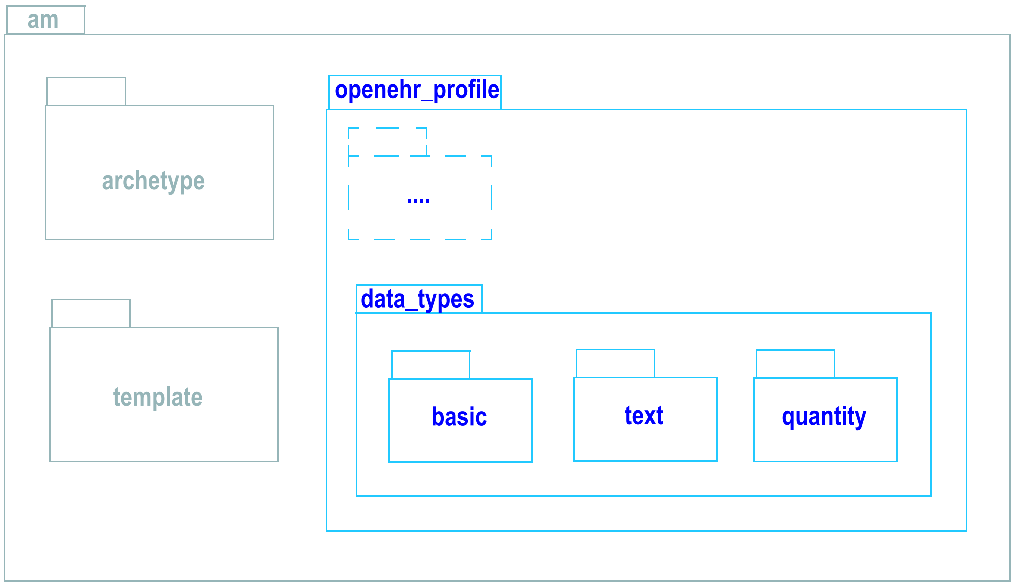

FIGURE 4 openehr.am.openehr_profile Package

3 Data_types.basic Package

The `am.openehr_profile.basic` package, illustrated in FIGURE 5, defines custom types for constraining the RM type `DV_STATE`.

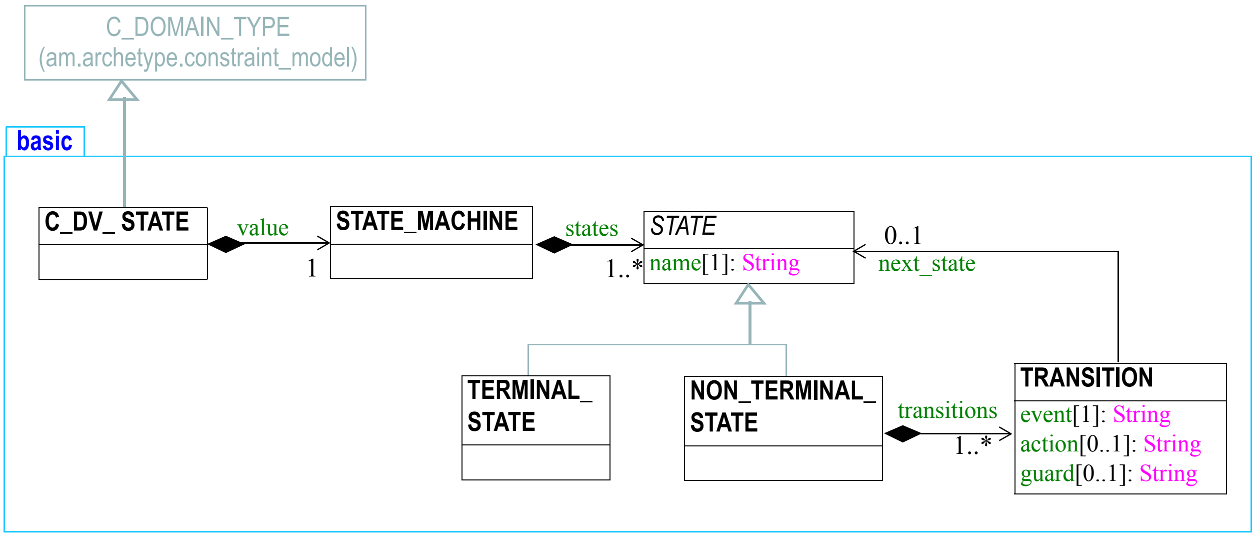

FIGURE 5 am.openehr_profile.data_types.basic Package

A example of a state machine to model the state of a medication order is illustrated in FIGURE 6. This state machine is defined by an instance of the class `STATE_MACHINE`. (Note that for general modelling of states of medications and other interventions, the standard state machine defined in the EHR IM should normally be used).

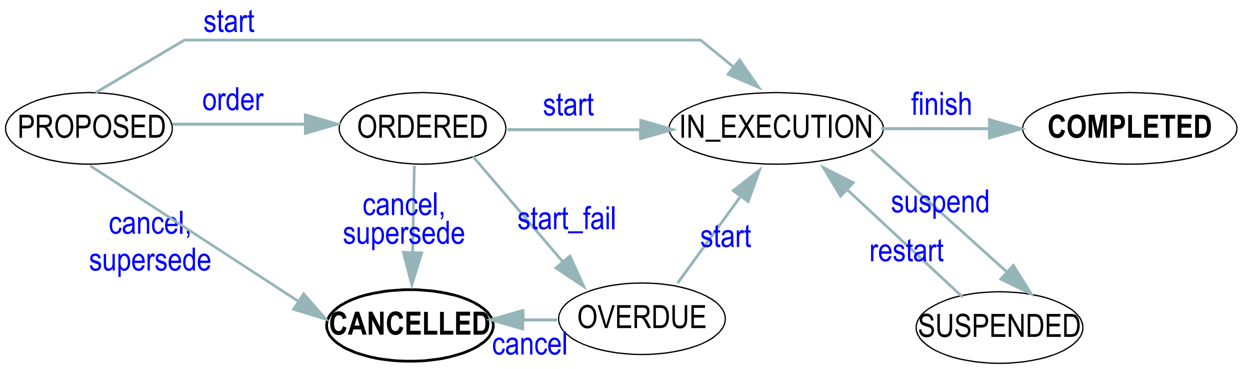

FIGURE 6 Example State Machine for Medication Orders

3.1 Class Descriptions

3.1.1 C_DV_STATE Class

CLASS	**C_DV_STATE**
Purpose	Constrainer type for `DV_STATE` instances. The attribute *c_value* defines a state/event table which constrains the allowed values of the attribute *value* in a `DV_STATE` instance, as well as the order of transitions between values.

CLASS	C_DV_STATE	
Inherit	`C_DOMAIN_TYPE`	
Attributes	Signature	Meaning
1..1	**value**: `STATE_MACHINE`	
Invariants	**value_exists**: c_value /= Void	

3.1.2 STATE_MACHINE Class

CLASS	STATE_MACHINE	
Purpose	Definition of a state machine in terms of states, transition events and outputs, and next states.	
Attributes	Signature	Meaning
1..1	**states**: `Set <STATE>`	
Invariants	***States_valid***: states /= Void **and then not** states.is_empty	

3.1.3 STATE Class

CLASS	*STATE (abstract)*	
Purpose	Abstract definition of one state in a state machine.	
Attributes	Signature	Meaning
1..1	**name**: `String`	name of this state
Invariants	***Name_valid***: name /= Void **and then not** name.is_empty	

3.1.4 NON_TERMINAL_STATE Class

CLASS	NON_TERMINAL_STATE	
Purpose	Definition of a non-terminal state in a state machine, i.e. one that has transitions.	
Inherit	`STATE`	
Attributes	Signature	Meaning
1..1	**transitions**: `Set <TRANSITION>`	
Invariants	***Transitions_valid***: transitions /= Void **and then not** transitions.is_empty	

3.1.5 TERMINAL_STATE Class

CLASS	TERMINAL_STATE	
Purpose	Definition of a terminal state in a state machine, i.e. a state with no exit transitions.	
Inherit	`STATE`	
Attributes	**Signature**	**Meaning**
Invariants		

3.1.6 TRANSITION Class

CLASS	TRANSITION	
Purpose	Definition of a state machine transition.	
Attributes	**Signature**	**Meaning**
1..1	**event**: `String`	Event which fires this transition
0..1	**guard**: `String`	Guard condition which must be true for this transition to fire
0..1	**action**: `String`	Side-effect action to execute during the firing of this transition
1..1	**next_state**: `STATE`	Target state of transition
Invariants	***Event_valid***: event /= Void **and then not** event.is_empty ***Action_valid***: action /= Void **implies not** action.is_empty ***Guard_valid***: guard /= Void **implies not** guard.is_empty ***Next_state_valid***: next_state /= Void	

4 Data_types.text Package

4.1 Overview

The `am.openehr_profile.data_types.text` package contains custom classes for expressing constraints on instances of the types defined in the rm.data_types.text package. Only one type is currently defined, enabling the constraining of `CODE_PHRASE` instances. It is illustrated in FIGURE 7.

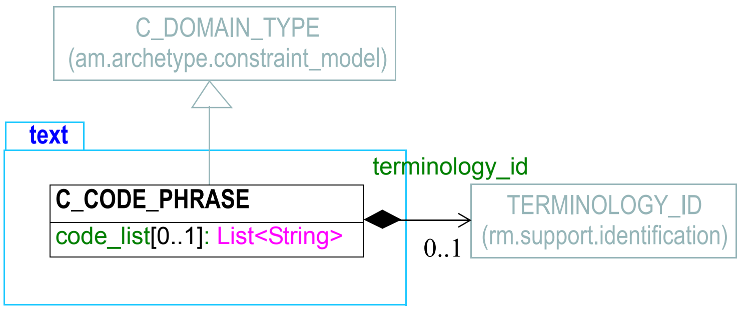
C_DOMAIN_TYPE
(am.archetype.constraint_model)
text
C_CODE_PHRASE
code_list[0..1]: List<String>
terminology_id
0..1
TERMINOLOGY_ID
(rm.support.identification)

FIGURE 7 am.openehr_profile.data_types.text Package

4.2 Requirements

The primary requirement of constraints on coded terms in archetypes is to be able to state a logical constraint which does not limit the archetype to only being used with one particular vocabulary; in other words that *constraints on codes not limit the (re)usability of the archetype*. With respect to object models of data, the requirements for constraints on coded terms relate to their use as names and as values.

Constraints on Names

Where coded names occur in data e.g. in instances of `FOLDER`.*name*, `SECTION`.*name*, and `CLUSTER`.*name*, the following types of constraints are needed:

- require the term to be a particular one from a particular terminology, e.g. the ICD10 term "diabetes mellitus" (here the terminology is not limited to one value set);
- require the term to be any term from a particular terminology constrained by some relationship within the terminology, e.g. "is-a"; for example, "any term in ICD10 which is-a 'tropical infection'";
- require the term to be any term from a particular terminology, e.g. the HL7 PracticeSetting domain (here the terminology itself is limited to one value set);

Constraints on Values

The second kind of constraint on coded terms is used where terms appear as values. In this case, the intention is to specify a set of allowed terms, for example blood groups, diagnoses which may be relevant in the particular clinical setting, or the characteristics of a lump on palpation. More complex constraints specify that the set of terms is the union of two or more groups (the OR operator in queries), or is a member of a number of groups (the AND operator in queries), or even some more complex combination. In all cases, we can think of the constraint as returning a "candidate set of terms" when evaluated against real terminologies.

A candidate set of terms can be obtained from a terminology in a number of ways. First, via the use of *relationships* encoded in the terminology, such as: "X is-a-kind-of coronary disease", where classification relationships such as “is-a-kind-of” are defined in the terminology of interest. Second, by identifying terms which belong in some kind of group or category. Consider a constraint such as “X has-category palpable-body-part” which will return the set of terms which describe palpable body parts. These two methods may be mixed as in “X is-a-kind-of body-part AND has-category palpable”, which uses both a relationship and a category - and is equivalent to the previous category described. Note that a constraint like “X is-a-kind-of body-part” is likely to return a long list of body parts, while the category of “palpable” body-parts would reduce this significantly. Such constraints should only be specified if there is a mechanism to implement the categorisation - this might not be in the terminology but must be available to the terminology service.

Further constraining can be achieved by the use of boolean relationships between candidate sets produced by the method above, however it should always be understood that every time this is done, it in some sense usurps the role of knowledge / terminology. In theory only terminologies and ontologies can say that more than one candidate set of terms can be meaningfully intersected (AND operator) or unioned (OR operator) to produce a final meaningful set. However, the current reality is that very few terminologies implement even a small percentage of the possible knowledge relationships, and such constraints will indeed need to be made inside some other part of the knowledge environment.

An example of such a constraint is:

```
X is-a ‘surface body region’ OR (X is-a ‘organ’ AND has-category ‘palpable’)
```

The general case for value sets of coded terms is nested boolean expressions, where each expression element is one of the following:

- a particular term
- a named relationship
- a named category

For such expressions to be safe, all terms, relationships and categories must come from the same version of the same terminology, or an intentionally designed adjunct to it. This is the only way that *intended* meanings can be accessed. To arbitrarily mix terms and relationships from different terminologies is effectively side-stepping the known semantics of each of the systems, and creating value sets based on semantics not defined by anyone.

4.3 Design

4.3.1 Standard ADL Approach

The generic kind of constraint that can be expressed for the `DV_CODED_TEXT` type can, like all standard archetype constraints, only include constraints on the attributes defined in the reference model type. This is illustrated by the following fragment of ADL:

```
DV_CODED_TEXT matches {
    defining_code matches {
        CODE_PHRASE matches {
            terminology_id matches {“xxxx”}
            code_string matches {“cccc”}
        }
    }
}
```

The standard approach allows the attributes *terminology_id* and *code_string* to be constrained independently, and would for example, allow *terminology_id* to be constrained to `ICD10|Snomed-ct|LOINC`, while *code_string* could be constrained to some particular fixed values. However, this make no sense; codes only make sense within a given terminology, not across them. It also makes no sense to allow codes from more than one terminology, as terminologies generally have quite different designs - LOINC and Snomed-CT are completely different in their conception and realisation.

4.3.2 Terminology-specific Code Constraints

A more appropriate kind of constraint for `CODE_PHRASE` instances is for *terminology_id* to be fixed to one particular terminology, and for *code_string* to be constrained to a set of allowed codes; an empty list indicates that any code is allowed. These semantics are formalised in the class definition, shown below. The following examples, expressed in the dADL data language, illustrate instances of `C_CODE_PHRASE` expressing terminology-specific constraints.

-
  ```
  terminology_id = <“ICD10”>
  code_list = <[F43.1]> -- post traumatic stress disorder
  ```

-
  ```
  terminology_id = <“ICD10”>
  subset = <[xxx]> -- acute stress reactions
  code_list = <
  [F43.00], -- acute stress reaction, mild
  [F43.01], -- acute stress reaction, moderate
  [F32.02] -- acute stress reaction, severe
  >
  ```

-
  ```
  terminology_id = <“SNOMED-CT”>
  subset = <[xxx]> -- body structures
  ```

4.3.3 Terminology-neutral Code Constraints

The above approach to constraining term codes is only applicable when the particular terminology mentioned in the constraint is really the only sensible one for the purpose, and would not compromise the reusability of the archetype by the widest possible audience. It may be reasonable to constrain a value field in a particular archetype to e.g. an ICD10 code for “chronic obstructive pulmonary disease (COPD)”; this may be accepted globally as the right thing to do (given that one can reasonably call ICD10 a terminology of global availability and applicability). However, using e.g. LOINC codes for lab analyte names might not be appropriate - it may be accepted in the US and other countries using LOINC for laboratory result encoding, but probably not elsewhere.

A more sophisticated way of constraining codes is therefore needed for this situation. This can be done in three ways:

- defining coded terms inside the archetype itself - i.e. treating the archetype as a micro-vocabulary;
- without referring to any vocabulary at all (and assuming that the binding to a particular vocabulary would be done at some other place in the computing environment);
- or by allowing bindings to multiple vocabularies/terminologies to be explicitly stated somewhere in the archetype.

Archetype-local Codes

A relatively simple of way of using particular coded terms in the archetype, while guaranteeing that the archetype is re-usable is simply to define such terms in the archetype ontology and use them. This treats the archetype as a small vocabulary in its own right, and avoids the problem of the mess of terminologies in the real world.

The following ADL examples illustrate the use of archetype-local coded terms:

```
    defining_code matches {[local::at0016]}
```

```
    defining_code matches {[hl7_ClassCode::EVN, OBS]}

    defining_code matches {
        [local::
           at1311, -- Colo-colonic anastomosis
           at1312, -- Ileo-colonic anastomosis
           at1313, -- Colo-anal anastomosis
           at1314, -- Ileo-anal anastomosis
           at1315] -- Colostomy
    }
```

These can all be represented as instances of the class `C_CODE_PHRASE` by simply setting terminology_id to "local".

Abstract Inline Queries

The second approach above implies some kind of abstract terminology query language. Currently, no definitive language for this purpose exists, although there is research in this area. The `C_CODE_PHRASE` model above accommodates this as a future possibility, with the *query* attribute, which would allow a query to some service to be expressed.

External bindings in the Archetype Ontology

The third approach above is already provided for in archetypes, via the use of "ac" coded nodes referring to concrete queries to particular terminologies, stored in the archetype ontology section. An equivalent query can be expressed for any number of terminologies by this method. Nothing is needed in the `C_CODE_PHRASE` type to support this, since a `CONSTRAINT_REF` object is used instead (see the *open*EHR AOM). An example in ADL of the use of "ac" codes is:

```
    defining_code matches {[ac0016]}-- type of respiratory illness
    property matches {[ac0034]} -- acceleration
```

Here, the `[acNNNN]` codes might refer to queries into a terminology and units service, respectively, such as the following (in dADL):

```
    items("ac0016") = <query("terminology", "terminology_id = ICD10AM and ...")
    items("ac0034") = <query("units", "X matches 'DISTANCE/TIME^2'")
```

4.4 Pre-evaluation

An archetype containing instances of `C_CODE_PHRASE` could be evaluated in advance against a terminology, to generate the actual sets of candidate terms, allowing the populated archetype to be distributed and used for coding even by sites without access to coding systems.

4.5 Class Descriptions

4.5.1 C_CODE_PHRASE Class

CLASS	C_CODE_PHRASE
Purpose	Express constraints on instances of `CODE_PHRASE`. The *terminology_id* attribute may be specified on its own to indicate any term from a specified terminology; the *code_list* attribute may be used to limit the codes to a specific list.
Inherit	`C_DOMAIN_TYPE`

CLASS	C_CODE_PHRASE	
Attributes	**Signature**	**Meaning**
0..1 (cond)	**terminology_id**: `TERMINOLOGY_ID`	Syntax string expressing constraint on allowed primary terms
0..1 (cond)	**code_list**: `List<String>`	List of codes; may be empty
Invariants	***List_validity***: code_list /= Void **implies** (**not** code_list.is_empty **and** terminology_id /= Void) ***Any_allowed_validity***: any_allowed = (terminology_id = Void **and** code_list = Void)	

5 Data_types.quantity Package

5.1 Overview

The am.openehr_profile.data_types.quantity package is illustrated in FIGURE 8. Two custom types are defined: C_DV_QUANTITY and C_DV_ORDINAL.

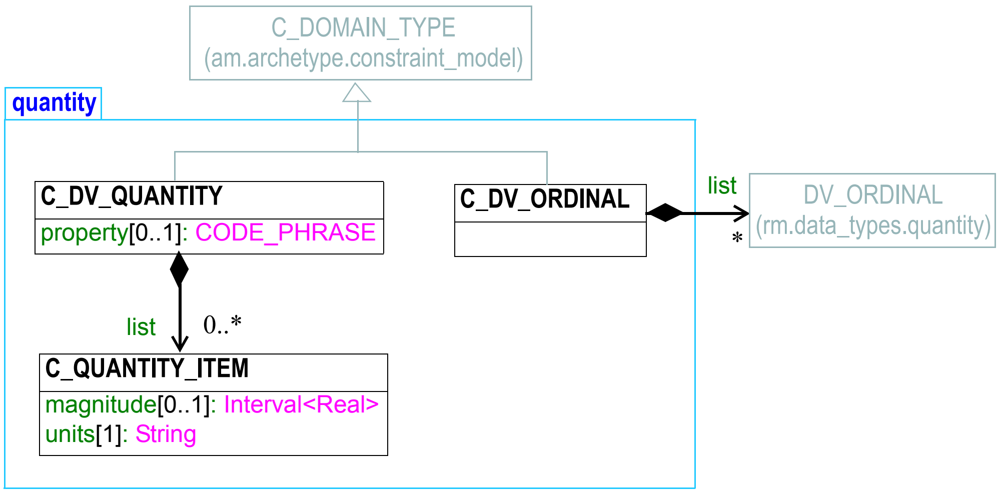

FIGURE 8 am.openehr_profile.data_types.quantity Package

5.2 Design

5.2.1 Ordinal Type Constraint

An ordinal value is defined as one which is ordered without being quantified, and is represented by a symbol and an integer number. The DV_ORDINAL class can be constrained in a generic way in ADL as follows:

```
item matches {
   ORDINAL matches {
      value matches {0}
      symbol matches {
         CODED_TEXT matches {
            code matches {[local::at0014]} -- no heartbeat
         }
      }
   }
   ORDINAL matches {
      value matches {1}
      symbol matches {
         CODED_TEXT matches {
            code matches {[local::at0015]} -- less than 100 bpm
         }
      }
   }
   ORDINAL matches {
      value matches {2}
      symbol matches {
         CODED_TEXT matches {
            code matches {[local::at0016]} -- greater than 100 bpm
```

```
                        }
                    }
                }
            }
```

The above says that the allowed values of the attribute value is the set of ORDINALs represented by three alternative constraints, each indicating what the numeric value of the ordinal in the series, as well as its symbol, which is a CODED_TEXT.

A more efficient way of representing the same constraint is using the following ADL syntax:

```
    item matches {0|[local::at0014], 1|[local::at0015], 2|[local::at0016]}
```

In the above expression, each item in the list corresponds to a single ORDINAL, and the list corresponds to an implicit definition of an ORDINAL type, in terms of the set of its allowed values. The object equivalent of this syntax is given by the custom class C_DV_QUANTITY, which efficiently allows a DV_QUANTITY to be constrained in terms of a set of DV_ORDINALs.

5.3 Class Definitions

5.3.1 C_DV_ORDINAL Class Definition

CLASS	**C_DV_ORDINAL**	
Purpose	Class specifying constraints on instances of DV_ORDINAL. Custom constrainer type for instances of DV_ORDINAL.	
Inherit	C_DOMAIN_TYPE	
Attributes	**Signature**	**Meaning**
1..1	**list**: Set<DV_ORDINAL>	Set of allowed DV_ORDINAL values.
Invariants	***Ordinals_valid***: items /= Void **xor** any_allowed ***Items_valid***: items /= Void **implies not** items.is_empty	

5.3.2 C_DV_QUANTITY Class Definition

CLASS	**C_DV_QUANTITY**	
Purpose	Constrain instances of DV_QUANTITY.	
Inherit	C_DOMAIN_TYPE	
Attributes	**Signature**	**Meaning**
0..1	**list**: List<C_QUANTITY_ITEM>	List of value/units pairs.
0..1	**property**: CODE_PHRASE	Optional constraint on units property

CLASS	C_DV_QUANTITY
Invariants	***Items_valid***: list /= Void **implies not** list.is_empty ***Property_valid***: property /= Void **implies** terminology(Terminology_id_openehr).has_code_for_group_id (Group_id_measurable_properties, property) ***Overall_validity***: (list /= Void **or** property /= Void) **xor** any_allowed

5.3.3 C_QUANTITY_ITEM Class Definition

CLASS	C_QUANTITY_ITEM	
Purpose	Constrain instances of `DV_QUANTITY`.	
Attributes	Signature	Meaning
0..1	**magnitude**: `Interval<Real>`	Value must be inside the supplied interval.
1..1	**units**: `STRING`	Constraint on units
Invariants	***units_valid***: units /= Void **and not** units.is_empty	

END OF DOCUMENT