Archetype Definition Language (ADL)

Editors:{T Beale, S Heard}[1]

Revision: 1.1

Pages: 69

Keywords: EHR, health records, modelling, constraints, software

1. Ocean Informatics Australia

Copyright Notice

Amendment Record

Issue	Details	Who	Completed
1.1	CR-000079. Change interval syntax in ADL.	T Beale	24 Jan 2004
1.0	CR-000077. Add cADL date/time pattern constraints. CR-000078. Add predefined clinical types. Better explanation of cardinality, occurrences and existence.	S Heard, T Beale	14 Jan 2004
0.9.9	CR-000073. Allow lists of Reals and Integers in cADL. CR-000075. Add predefined clinical types library to ADL.	T Beale, S Heard	28 Dec 2003
0.9.8	CR-000070. Create Archetype System Description. Moved Archetype Identification Section to new Archetype System document. Copyright Assgined by Ocean Informatics P/L Australia to The *open*EHR Foundation.	T Beale, S Heard	29 Nov 2003
0.9.7	Added simple value list continuation (“,...”). Changed path syntax so that trailing ‘/’ required for object paths. Remove ranges with excluded limits. Added terms and term lists to dADL leaf types.	T Beale	01 Nov 2003
0.9.6	Additions during HL7 WGM Memphis Sept 2003	T Beale	09 Sep 2003
0.9.5	Added comparison to other formalisms. Renamed CDL to cADL and dDL to dADL. Changed path syntax to conform (nearly) to Xpath. Numerous small changes.	T Beale	03 Sep 2003
0.9	Rewritten with sections on cADL and dDL.	T Beale	28 July 2003
0.8.1	Added basic type constraints, re-arranged sections.	T Beale	15 July 2003
0.8	Initial Writing	T Beale	10 July 2003

Table of Contents

1 Introduction

1.1 Purpose

This document describes the syntax and design basis of the Archetype Definition Language (ADL). It is intended for software developers, and technically-oriented domain specialists and subject matter experts (SMEs). Although ADL is primarily intended to be read and written by tools, it is quite readable by humans and ADL archetypes can be hand-edited using a normal text editor.

1.2 Overview

1.2.1 What is ADL?

Archetype Definition Language (ADL) is a formal language for expressing archetypes, which are constraint-based models of domain entities, or what some might call "structured business rules". The archetype concept is described in [1], [2], [3], and [4]. ADL uses two other syntaxes, cADL (constraint form of ADL) and dADL (data definition form of ADL) to describe constraints on data which are instances of some information model (e.g. expressed in UML). It is most useful when very generic information models are used for describing the data in a system, for example, where the logical concepts `PATIENT`, `DOCTOR` and `HOSPITAL` might all be represented using a small number of classes such as `PARTY` and `ADDRESS`. In such cases, archetypes are used to constrain the *valid* structures of instances of these generic classes to represent the desired domain concepts. In this way future-proof information systems can be built - relatively simple information models and database schemas can be defined, and archetypes supply the semantic modelling, completely outside the software. ADL can thus be used to write archetypes for any domain where formal object model(s) exist which describe data instances.

When archetypes are used at runtime in particular contexts, they are composed into larger constraint structures, with local or specialist constraints added, by the use of *templates*. The formalism of templates is the template form of ADL (tADL) (to be described). Archetypes can also be specialised.

1.2.2 Relationship to Other Information Artifacts

Archetypes are distinct, structured models of domain concepts, such as "blood pressure", and sit between lower layers of knowledge resources in a computing environment, such as clinical terminologies and ontologies, and actual data in production systems. Their primary purpose is to provide a way of managing generic data so that it conforms to particular domain structures and semantic constraints. Consequently, they bind terminology and ontology concepts with information model semantics, to make statements about what valid dat structures look like. ADL provides a solid formalism for expressing, building and using these entities computationally.

1.2.3 Structure

Archetypes expressed in ADL resemble programming language files, and have a defined syntax. ADL itself is a very simple "glue" syntax, which uses two other well-defined syntaxes for expressing structured constraints and data, respectively. The cADL syntax is used to express the archetype `definition`, while the dADL syntax is used to express data, which appears in the `description` and `ontology` sections of an ADL archetype. The top-level structure of an ADL archetype is shown in FIGURE 1.

This main part of this document describes dADL, cADL before going on to describe ADL, archetypes and domain-specific type libraries and syntax.

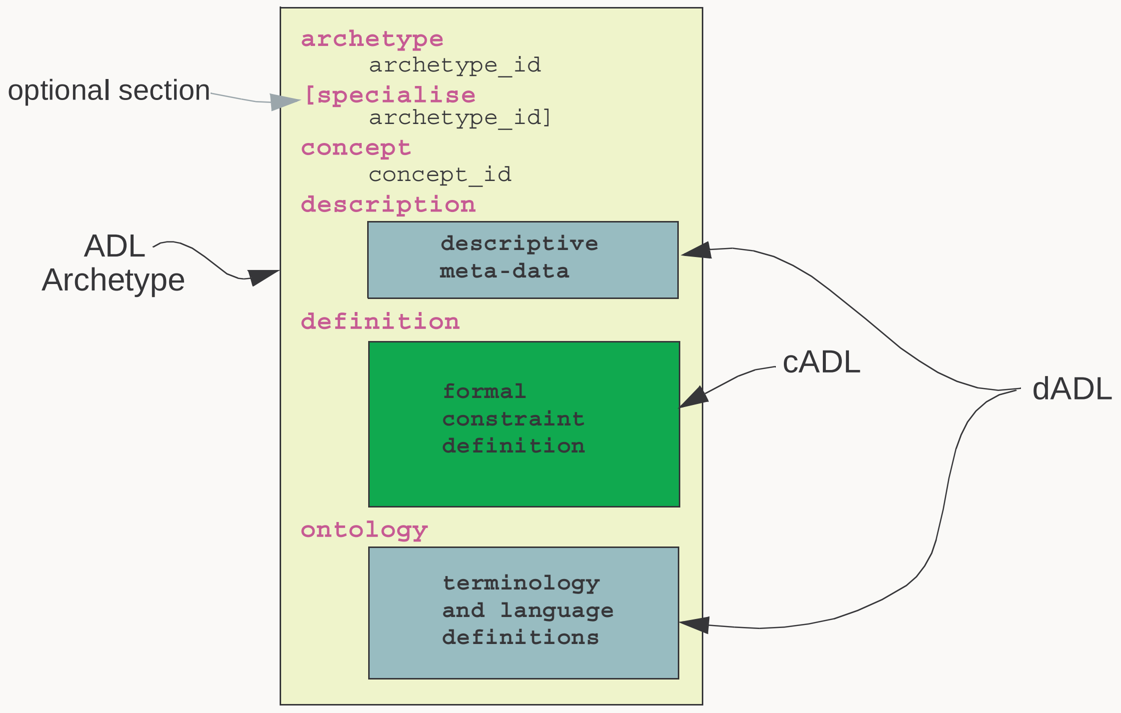


FIGURE 1 ADL Archetype Structure

1.2.4 An Example

The following is an example of a very simple archetype, giving a feel for the syntax. The main point to glean from the following is that the notion of ‘guitar’ is defined in terms of *constraints* on a *generic* model of the concept INSTRUMENT. The names mentioned down the left-hand side of the definition section (“INSTRUMENT”, “size” etc) are alternately class and attribute names from an object model. Each block of braces encloses a specification for some particular set of instances that conform to a specific concept, such as ‘guitar’ or ‘neck’, defined in terms of constraints on types from a generic class model. The leaf pairs of braces enclose constraints on primitive types such as Integer, String, Boolean and so on.

```
archetype
    adl-test-instrument.guitar.draft

concept
    [at0000] -- guitar

definition
    INSTRUMENT[at0000] matches {
       size matches {60..120}                   -- assumed in cm
       date_of_manufacture matches {yyyy-mm-??}  -- year & month ok
       parts cardinality matches {0..*} matches {
          PART[at0001] matches {                -- neck
             material matches {[local::at0003]}  -- timber
          }
          PART[at0002] matches {                -- body
             material matches {[local::at0003,
                                 at0004]}      -- timber or steel
          }
       }
```

```
         }

    ontology
        primary_language = <"en">
        languages_available = <"en", ...>

        term_definitions("en") = <
           items("at0000") = <
              text = <"guitar">;
              description = <"stringed instrument">
           >
           items("at0001") = <
              text = <"neck">;
              description = <"neck of guitar">
           >
           items("at0002") = <
              text = <"timber">;
              description = <"straight, seasoned timber">
           >
           items("at0003") = <
              text = <"steel">;
              description = <"stainless steel">
           >
           items("at0004") = <
              text = <"body">;
              description = <"body of guitar">
           >
        >
```

1.2.5 Relationship to Object Models

As a parsable syntax, ADL has a formal relationship with structural models such as those expressed in UML, according to the scheme of FIGURE 2. Here we can see that ADL documents are parsed into a network of objects (often known as a 'parse tree') which are themselves defined by a formal, abstract object model. Such a model can in turn be re-expressed as any number of concrete models, such as in a programming language, XML-schema or OMG IDL.

However, there can also be more than one abstract object model whose objects could be generated by an ADL parser, based on exactly the same input. One reason for this would be to define an information model (IM) independent ADL class model, and various IM-dependent ones. In the former, all ADL object nodes would be represented using instances of the same class, say ADL_NODE, whereas in the latter, they would be instances of different classes - an ADL SECTION node might be parsed to an instance of a class C_SECTION, which expresses constraints on instances of the class SECTION, found in for example, the CEN 13606 EHR model, the HL7 CDA model, and the *open*EHR EHR reference model. The IM-independent class model can be considered a direct transform of the ADL EBNF specification, and is described in this document in the sections dADL Object Model on page 23 and cADL Object Model on page 42.

Regardless of the possible object models, the ADL syntax remains constant as the primary formalism for authoring and sharing archetypes. A further serialised form of ADL is possible, by translating the ADL EBNF into its own XML language, i.e. writing an XML DTD directly for ADL, as indicated on the diagram by "XADL". If this is in fact possible, XADL archetypes would be completely equivalent to ADL archetypes, and could also be used as the medium of authoring and/or sharing. This document describes only standard ADL, but it can be assumed that all the semantics described would exist in a notional XADL, the only difference being in the types of tools which would be used with each form.

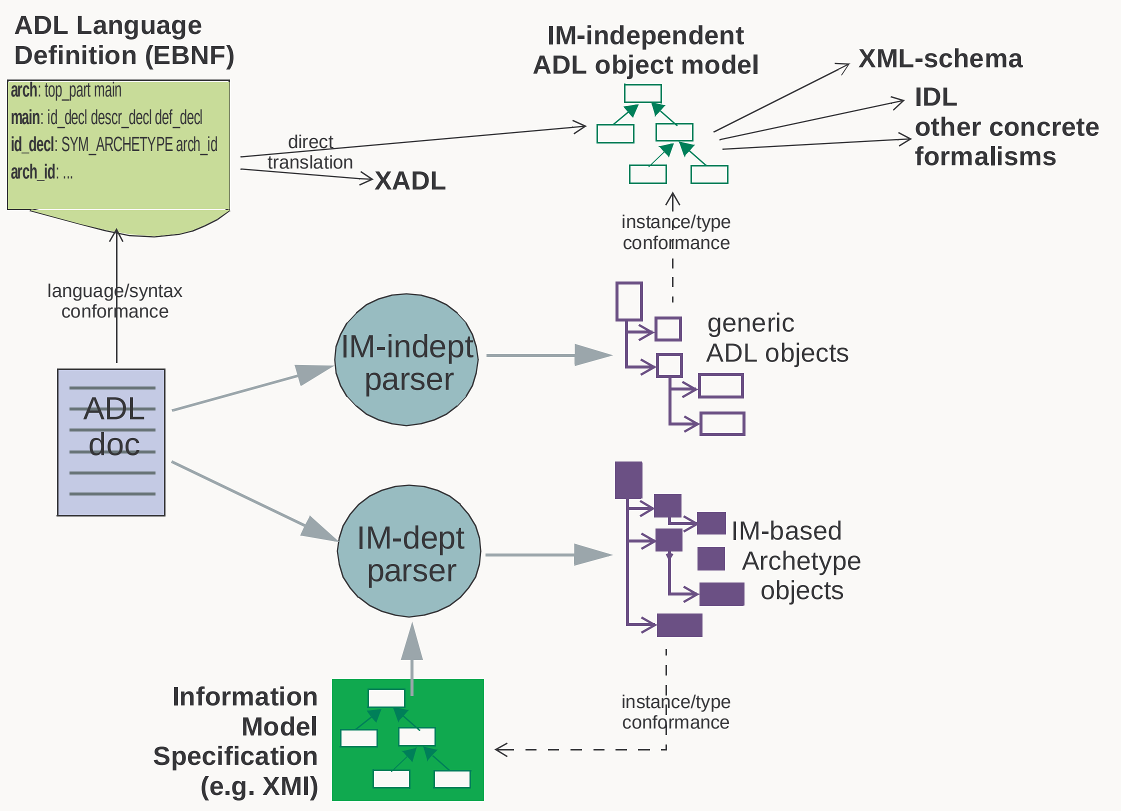

ADL Language Definition (EBNF)
arch: top_part main
main: id_decl descr_decl def_decl
id_decl: SYM_ARCHETYPE arch_id
arch_id: ...
direct translation
XADL
IM-independent ADL object model
XML-schema
IDL
other concrete formalisms
instance/type conformance
language/syntax conformance
IM-indept parser
generic ADL objects
ADL doc
IM-dept parser
IM-based Archetype objects
instance/type conformance
Information Model Specification (e.g. XMI)

FIGURE 2 Relationship of ADL with Object Models

1.3 Relationship to Other Formalisms

Whenever a new formalism is defined, it is reasonable to ask the question: are there not existing formalisms which would do the same job? The following sections compare ADL to other formalisms and show why it is different.

1.3.1 XML

Although, as pointed out above, it is possible to define an XADL language, the first priority was to develop an abstract syntax without the interference of other syntaxes for concrete data representation. ADL has been developed without using XML in the first instance because it simplifies the syntax definition, and ensures that none of the shortcomings of XML obstruct the needs of archetype expression as they occasionally have in the past (in the case of XADL for example, the limitations of DTD of XML-schema could come into play). In essence, ADL allows the language developer to deal with only one formalism (ADL), not two (ADL and XML). This is the same strategy as embodied in the abstract OWL syntax compared to its concrete RDF-based syntax (see below). ADL is also human readable, whereas XML is only useful for machine to machine communication (although some tortured souls continue to believe that XML, because it can be read into a text editor, is somehow intended directly for humans...). Further it provides a formal terminology-based node identification mechanism, and based on that, a path language.

1.3.2 XML-schema

Previously, archetypes have been expressed as XML instance, conforming to W3C XML schemas, for example in the Good Electronic Health Record (GeHR; see http://www.gehr.org) and *open*EHR projects. The schemas used in those projects correspond technically to the XML expressions of IM-dependent object models shown in FIGURE 2. XML archetypes are accordingly equivalent to serialised instances of the parse tree, i.e. particular ADL archetypes serialised from objects into XML instance.

With ADL parsing tools it is possible to convert ADL to any number of forms, including various XML formats, offering greater flexibility than previously. In this role, all XML models of archetype semantics are treated as derivations of ADL syntax; correspondingly, archetypes expressed in XML are treated as being derived from ADL.

1.3.3 OWL (Web Ontology Language)

The Web Ontology Language (OWL) [14] is a W3C initiative for defining ontologies in a form which can be used on the web, and is formally an extension of RDF (Resource Description Framework). OWL is a general purpose ontology language, and is primarily used to describe "classes" of things in such a way as to support inferencing on data. There is in general no assumption that the data itself was built based on any particular class model - it might be audio-visual objects in an archive, technical documentation for an aircraft or the web pages of a company. OWL's "class" definitions can usually be considered as constraint statements on an implied model on which data appears to be based. Restrictions are a primary way of defining classes. For example, the following fragment uses a property restriction to say that the class `Opera` (a subclass of `owl:Class`) has at least one librettist.

```
<owl:Class rdf:about="#Opera">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasLibrettist" />
      <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Possible ADL equivalents include:

```
DOCUMENT ∈ {
    class ∈ {[ac0001]}                  -- resolves to "Score"
    type ∈ {[ac0002]}                   -- resolves to "Opera"
    attributes cardinality ∈ {0..*} ∈ {
       DOC_ATTR occurrences ∈ {1..*} ∈ {
          name ∈ {ac0003}               -- resolves to "librettist"
          value ∈ {*}
       }
    }
}
```

and:

```
OPERA ∈ {
    librettist cardinality ∈ {1..*} ∈ {*}
}
```

The first version is targetted to data defined by a generic class model containing the class `DOCUMENT` with attributes *class* and *type*, and the class `DOC_ATTR` with attributes *name* and *value*, while the second is targetted to a model which contains the class `OPERA` with attribute *librettist*.

While appearing to be semantically equivalent, there are subtle differences in the OWL and ADL approaches in these examples. The OWL fragment is essentially defining the class `Opera` as "all those resources for which the property `hasLibrettist` exists, and has one or more values" - in other words a constraint which has to be interpreted by a query engine interrogating a repository of electronic documents. How the query engine works might be complex, depending on how few assumptions are able to be made about the content’s form. The ADL fragments both state constraints on data built according to known data models, and from the point of view of querying, it is a simple exercise to find matches. Seen as query objects, ADL archetypes and OWL ontologies might be able to be proven equivalent. However, one of the primary purposes of archetypes is to be used in creating data in the first place, and this is where the in-built assumptions of what object model an archetype is targetted to are crucial. It is not yet clear how a general purpose OWL class could be used for this purpose, although it may be possible.

Despite the differences, there appears to be a lot in common between OWL and ADL, and an initial review of semantics reveals similar coverage. It may be that the only significant differences are a) that ADL is in its pure form a non-XML syntax, and therefore unencumbered by any of the limitations of the XML syntaxes, b) that ADL archetypes are always written with respect to some object model, and c) that node identification and path processing is built in to ADL. Indeed one of the main justifications of ADL is to clearly and precisely described the desired semantics of constraint on object models, without reference to other syntaxes, in order to further the research on archetypes and templates; this is not intended to discount the future use of other epxressions or tools, and it may be that in the future tools can be written to enable ADL to be fully convertible to OWL and vice-versa, enabling the same archetypes to be used in XML and non-XML environments (such as pure OO systems).

1.3.4 OCL (Object Constraint Language)

The OMG’s Object Constraint Language (OCL) appears to be an obvious contender for writing constraint definitions for object models. However, its real use is to write constraints *within* object models, including function pre- and post-conditions, and class invariants. Accordingly, there is no structural character to the syntax - all statements are essentially first-order predicate logic statements about elements in models expressed in UML. This makes it impossible to use OCL to describe an archetype in a structural way which is natural to domain experts. OCL also has some flaws [5].

However, OCL is in fact relevant to ADL. ADL archetypes include invariants (and one day, might include pre- and post-conditions). Currently these are expressed in a syntax very similar to OCL, with certain differences. The exact definition of the ADL invariant syntax in the future will depend somewhat on the progress of OCL through the OMG standards process.

As with OWL, it may be possible in the future to prove that any ADL archetype can be losslessly transformed to OCL statements. It should be noted that the predicate logic flavour of OCL is closer semantically and syntactically to the way many software developers think about constraints than any of the XML syntaxes.

1.3.5 KIF (Knowledge Interchange Format)

The Knowledge Interchange Format (KIF) is a knowledge representation language whose goal is to be able to describe formal semantics which would be sharable among software entities, such as information systems in an airline and a travel agency. An example of KIF (taken from [8]) used to describe the simple concept of “units” in a `QUANTITY` class is as follows:

```
(defrelation BASIC-UNIT
     (=> (BASIC-UNIT ?u) ; basic units are distinguished
        (unit-of-measure ?u))) ; units of measure

(deffunction UNIT*
     ; Unit* maps all pairs of units to units
     (=> (and (unit-of-measure ?u1)
               (unit-of-measure ?u2))
        (and (defined (UNIT* ?u1 ?u2))
                 (unit-of-measure (UNIT* ?u1 ?u2))))
    ; It is commutative
    (= (UNIT* ?u1 ?u2) (UNIT* ?u2 ?u1))
    ; It is associative
    (= (UNIT* ?u1 (UNIT* ?u2 ?u3))
        (UNIT* (UNIT* ?u1 ?u2) ?u3)))

    (deffunction UNIT^
        ; Unit^ maps all units and reals to units
        (=> (and (unit-of-measure ?u)
              (real-number ?r))
           (and (defined (UNIT^ ?u ?r))
                 (unit-of-measure (UNIT^ ?u ?r))))
    ; It has the algebraic properties of exponentiation
    (= (UNIT^ ?u 1) ?u)
    (= (unit* (UNIT^ ?u ?r1) (UNIT^ ?u ?r2))
        (UNIT^ ?u (+ ?r1 ?r2)))
    (= (UNIT^ (unit* ?u1 ?u2) ?r)
        (unit* (UNIT^ ?u1 ?r) (UNIT^ ?u2 ?r)))
```

It should be clear from the above that KIF is a definitional language - it defines all the concepts it mentions. This is not the situation for which ADL was designed. The most common situation in which we find ourselves is that information models already exist, and may even have been deployed as software. The goal of ADL is to express constraints on instances of those models; thus an ADL archetype constraining QUANTITY objects will always be able to assume that a semantic definition of UNIT also exists outside of ADL. Thus, ADL itself would not be used to define the semantics of QUANTITY.*units*, rather it will constrain the definition of the same in some existing object model.

1.3.6 Schematron

To Be Continued:

1.4 Tools

A validating ADL parser is freely available from http://www.openEHR.org. The EBNF production rules for the parser and lexical specification are also available on the website.

2 dADL - Data ADL

2.1 Overview

The dADL syntax provides a formal means of expressing *instance data* based on an underlying information model, which is both readable by humans and by machines. The general appearance is exemplified by the following:

```
PERSON[01234] = <
    name = <                                        -- person's name
       forenames =    <"Sherlock">
       family_name =  <"Holmes">
       salutation =   <"Mr">
    >
    address = <                                     -- person's address
       habitation_number = <"22a">
       street_name =  <"Baker St">
       city =         <"London">
       country =      <"England">
    >
>
```

In the above the identifiers `PERSON`, `name`, `address` etc are all assumed to come from an information model. The basic design principle of dADL is to be able to represent data in a way that is both machine-processible and human readable, while making the fewest assumptions possible about the information model to which the data conforms. To this end, type names are only used (optionally) for root nodes; otherwise, only attribute names and values are explicitly shown; no syntactical assumptions are made about whether the underlying model is relational, object-oriented or what it actually looks like. More than one information model can be compatible withe the same dADL-expressed data. Literal leaf values are only of 'standard' widely recognised types, i.e. Integer, Real, Boolean, String, Character and a range of Date/time types. In standard dADL, documented in this section, all other more sophisticated types are expressed structurally, with leaf values of these primitive types. Other domain-specific literal types are documented in Predefined Type Libraries on page 59.

2.2 Basics

2.2.1 Keywords

dADL has no keywords of its own - all identifiers are assumed to come from an information model.

2.2.2 Comments

In dADL, comments are indicated by the "--" characters. Multi-line comments are achieved using the "--" leader on each line where the comment continues. In this document, comments are shown in brown.

2.2.3 Quoting

The backslash character ('\') is used to quote reserved characters in dADL, which include '<', '>', and '"'. The only characters which need to be quoted inside a string are the double quote character ('"') and the backslash character itself.

2.2.4 Information Model Identifiers

Two types of identifiers from information models are used in dADL: type names and attribute names. Type names are shown in this document in all uppercase, e.g. `PERSON`, while attribute names are shown in all lowercase, e.g. `home_address`. In both cases, underscores are used to represent word breaks. This convention is used to maximise the readability of this document, and other conventions may be used, such as the common programmer's mixed-case convention exemplified by `Person` and `homeAddress`. The convention chosen for any particular dADL document should be based on the convention used in the underlying information model. Identifiers are shown in green in this document.

2.2.5 Instance Identifiers

Data instances are identified in dADL using an identifier delimited by brackets, e.g. `[some_id]`. Any string may appear within the brackets, depending on how it is used. Instance identifiers may be used to identify and refer to data expressed in dADL, but also to external entities. Instance identifiers are shown in magenta.

```
To Be Continued:        some rules for this will be required
```

2.2.6 Semi-colons

Semi-colons can be used to separate dADL blocks, for example when it is preferable to include multiple attribute/value pairs on one line. Semi-colons make no semantic difference at all, and are included only as a matter of taste. The following examples are equivalent:

```
    items(2) = <text = <"plan">; description = <"The clinician's advice">>

    items(2) = <text = <"plan"> description = <"The clinician's advice">>

    items(2) = <
        text = <"plan">
        description = <"The clinician's advice">
    >
```

2.3 Structure

2.3.1 Content

The structure of dADL is purely hierarchical, and consists of the general pattern:

```
    attribute_id = <value>
```

The `attribute_id` may be an attribute name, such as `address`, or an attribute name followed by a qualifier, such as `addresses("home")`, `addresses("work")`. Qualifiers can be of any basic comparable type, and must follow the same syntactical rules described below for data of primitive types appearing in the value structure. Qualified attributes allow for lists and hash table data structures to be easily expressed, without making any assumptions about how they are actually represented in any language.

The `<value>` part of the pattern above may be any depth of nested repetitions of the same pattern, giving a typical structure like the following:

```
    attribute =  <
        attribute = <
           attribute(1)  = <leaf_value>
           attribute(2)  = <leaf_value>
        >
        attribute = <
```

```
            attribute = <
               attribute = <leaf_value>
            >
            attribute = <leaf_value>
         >
     >
```

2.3.1.1 Empty Sections

Empty sections are allowed at both internal and leaf node levels, enabling the author to express the fact that there is in some particular instance, no data for an attribute, while still showing that the attribute itself is expected to exist in the underlying information model. An empty section looks as follows:

```
     address = <>             -- person's address
```

Nested empty sections can be used.

2.3.2 Anonymous Objects

Data expressed in dADL within some other document or syntax (such as an ADL archetype) in the form shown above are *anonymous*, meaning that they have no overall identifier. This might be done because there is no need to identify the data (there is only one instance of it), or it is preferred to shown no type information, which would imply more about the underlying information model. Multiple "trees" of anonymous dADL data can be used in the one document, usually implying separate data objects, such as in the following example:

```
     people = <
         ...
     >
     places = <
         ...
     >
```

However, it is sometimes preferable to include more typing information rather then less, and to be able to refer to blocks of data. To do this, *identified* dADL is needed.

2.3.3 Identified Objects

Anonymous dADL is usually used to express the data of an instance of some formal type, such as PERSON, in the first example above. The syntax of an explicitly identified dADL fragment follows the general pattern:

```
     TYPE[id] =   <
         attribute = <
            ...
         >
         attribute = <
            ...
         >
     >
```

Providing the type and a unique instance identifier for a fragment of dADL makes the fragment addressable from elsewhere, including from other fragments of dADL. As a consequence, a number of useful ways of expressing data become possible.

2.3.3.1 Decomposed Data

An instance of a business type which is really composed of instances of smaller types can be represented in two ways in dADL. The first is as one large block, as follows:

```
     ORGANIZATION[acme_fireworks] = <
```

```
        name("ACME Fireworks and Explosives") = <
        divisions("sales") = <
           head = <"Huan Xiu">
           location = <
              ...
           >
        >
        divisions("special events") = <
           head = <"W E Coyote">
           location = <
              ...
           >
        >
    >
```

The second is as a decomposition into the constituent identified dADL blocks:

```
    ORGANIZATION[acme_fireworks] = <
        name("ACME Fireworks and Explosives") = <
        divisions("sales") = <UNIT[acme_fireworks:sales]>
        divisions("special events") = <UNIT[acme_fireworks:sales]
    >
    UNIT[acme_fireworks:sales] = <
        id = <"sales">
        head = <"Huan Xiu">
        location = <
           ...
        >
    >
    UNIT[acme_fireworks:special_events] = <
        id = <"special events">
        head = <"W E Coyote">
        location = <
           ...
        >
    >
```

2.3.3.2 Shared Objects

The decomposed form of dADL can be used to express sharing of an object by other objects, by the use of the same identifier in more than one place.

2.3.4 Scope of a dADL Document

Multiple trees of dADL data occurring one after the other, whether identified or anonymous, are considered to comprise one dADL document. Accordingly, node identifiers are assumed to be globally unique at least within the document, if not at some higher level.

2.4 Leaf Data

All dADL data eventually devolve to instances of the primitive types `String`, `Integer`, `Real`, `Double`, `String`, `Character`, various date/time types, lists of these types, and a few special types. dADL does not use type or attribute names for instances of primitive types, only manifest values, making it possible to assume as little as possible about type names and structures of the primitive types. In all the following examples, the manifest data values are assumed to appear immediately inside a leaf pair of angle brackets, i.e.

```
        some_attribute = <manifest value here>
```

2.4.1 Atomic Types

2.4.1.1 Character Data

Characters are shown in a number of ways. In the literal form, a character is shown in single quotes, as follows:

```
'a'
```

Special characters are expressed using the ISO 10646 or XML special character codes as described above. Examples:

```
'&ohgr;'      -- greek omega
```

2.4.1.2 String Data

All strings are shown in inverted commas, as follows:

```
"this is a string"
```

Quoting and line extension is done using the backslash character, as follows:

```
"this is a much longer string, what one might call a \"phrase\" or even \
a \"sentence\" with a very annoying backslash (\\) in it."
```

String data can be used to contain almost any other kind of data, which is intended to be parsed as some other formalism. Special characters (including the inverted comma and backslash characters) are expressed using the ISO 10646 or XML special character codes within single quotes. ISO codes are mnemonic, and follow the pattern `&aaaa;`, while XML codes are hexadecimal and follow the pattern `&#xHHHH;`, where `H` stands for a hexadecimal digit. An example is:

```
"a &isin; A"     -- prints as: a ∈ A
```

2.4.1.3 Integer Data

Integers are represented simply as numbers, e.g.:

```
25
300,000
29e6
```

Commas for breaking long numbers are optional.

2.4.1.4 Real Data

Real numbers are assumed whenever a decimal is detected in a number, e.g.:

```
25.0
3.1415926
6.023e23
```

2.4.1.5 Boolean Data

Boolean values can be indicated by the following values (case-insensitive):

```
True
False
```

2.4.1.6 Dates and Times

In dADL, all dates and times are expressed in ISO8601 form, which enables dates, times, date/times and durations to be expressed. Patterns for dates and times based on ISO 8601 include the following:

```
yyyy-MM-dd                         -- a date
hh:mm[:ss[.sss][Z]]                -- a time
yyyy-MM-dd hh:mm:ss[.sss][Z]       -- a date/time
```

where:

```
yyyy    = four-digit year
MM      = month in year
```

```
    dd     = day in month
    hh     = hour in 24 hour clock
    mm     = minutes
    ss.sss = seconds, incuding fractional part
    Z      = the timezone in the form of a '+' or '-' followed by 4 digits
              indicating the hour offset, e.g. +0930, or else the literal 'Z'
              indicating +0000 (the Greenwich meridian).
```

Durations are expressed using a string which starts with "P", and is followed by a list of periods, each appended by a single letter designator: "D" for days, "H" for hours, "M" for minutes, and "S" for seconds. Examples of date/time data include:

```
    1919-01-23                -- birthdate of Django Reinhardt
    16:35.04                  -- rise of Venus in Sydney on 24 Jul 2003
    2001-05-12 07:35:20+1000  -- timestamp on an email received from Australia
    P22D4H15M0S               -- period of 22 days, 4 hours, 15 minutes
```

2.4.2 Intervals of Ordered Primitive Types

Intervals of any ordered primitive type, i.e., Integer, Real, Date, Time, Date_time and Duration, can be stated using the following uniform syntax, where N, M are instances of any of the ordered types:

`|N..M|`	in the inclusive range where N and M are integers, or the infinity indicator;
`|<N|`	less than N;
`|>N|`	greater than N;
`|>=N|`	greater than or equal to N;
`|<=N|`	less than or equal to N;
`|N +/-M|`	interval of N ± M.

Examples of this syntax include:

```
    |0..5|                    -- integer interval
    |0.0..1000.0 |            -- real interval
    |08:02..09:10|            -- interval of time
    |>= 1939-02-01|           -- open-ended interval of dates
    |5.0 +/-0.5|              -- 4.5 - 5.5
```

2.4.3 Lists of Primitive Types

Data of any primitive type can occur singly or in lists, which are shown as comma-separated lists of item, all of the same type, such as in the following examples:

```
    "cyan", "magenta", "yellow", "black" -- printer's colours
    1, 1, 2, 3, 5                        -- first 5 fibonacci numbers
    08:02, 08:35, 09:10                  -- set of train times
```

No assumption is made in the syntax about whether a list represents a set, a list or some other kind of sequence - such semantics must be taken from an underlying information model.

Lists which happen to have only one datum are indicated by using a comma followed by a list continuation marker of three dots, i.e. "...", e.g.:

```
    "en", ...         -- languages
    "icd10", ...      -- terminologies
    [at0200], ...
```

2.5 Paths

Paths can be constructed to refer to any node in dADL data. The general form of the syntax is as follows:

```
[‘/’|object_id] attr_name [‘[’ object_id ‘]’] {‘/’ attr_name [‘[’ object_id
‘]’ ‘/’]}
```

Essentially paths consist of segments separated by slashes (‘/’), where each segment is an attribute name with optional object indentifier. A path either finishes in a slash, and identifies an object node, or finishes in a relationship name, and identifies a relationship node. The optional leading item may be slash or an object id, in the case of identified dADL; either indicates an absolute path; the path is relative if the first segment is an attribute name.

The following data tree and example paths illustrate the syntax:

```
term_definitions("en") = <
    items("at0000") = <
       text = <"apgar result">
       description = <"apgar result of newborn">
    >
    items("at0001") = <
       text = <"history">
       description = <"history">
    >
    ...
>

/                                 -- root object node
/term_definitions                 -- ‘term_definitions’ relation node
/term_definitions[en]/            -- object after first ‘=’
/term_definitions[en]/items       -- ‘items’ relationship node inside block
/term_definitions[en]/items[at0001]/ -- at0001 text/description block
/term_definitions[en]/items[at0001]/text/   -- at0001 text value
```

2.5.1 Comparison with Xpath

The syntax above is structurally very similar to that used in the Xpath query language. A few differences are worth pointing out. Xpath differentiates between “children” and “attributes” sub-items of an object due to the difference in XML between Elements (true sub-objects) and Attributes (tag-embedded primitive values). In ADL, as with any pure object formalism, there is no such distinction, and all subparts of any object are referenced in the manner of Xpath children; in particular, in the Xpath abbreviated syntax, the qualifier `child::` does not need to be used.

Secondly, in the Xpath abbreviated syntax, the expression:

```
items[1]
```

means “the first object in the ‘items’ children of the current object”. In ADL, the “[1]” is read not as an ordinal number, but as an identifier, or key - it is equivalent to the Xpath expression:

```
items[@id=1]
```

ADL does not distinguish attributes from children, and also assumes the `id` attribute. Thus, the following expressions are legal in ADL, but might not be in Xpath:

```
items[1]         -- the member of ‘items’ with key ‘1’
items[foo]       -- the member of ‘items’ with key ‘foo’
items[at0001]    -- the member of ‘items’ with key ‘at0001’
```

Since one would expect that simple numeric ids (‘1’, ‘2’, ‘3’, etc) would correspond with ordinal positions in a list, the first path will refer to the same object in Xpath and ADL.

2.6 dADL Object Model

FIGURE 3 illustrates the essentials of the dADL object model.

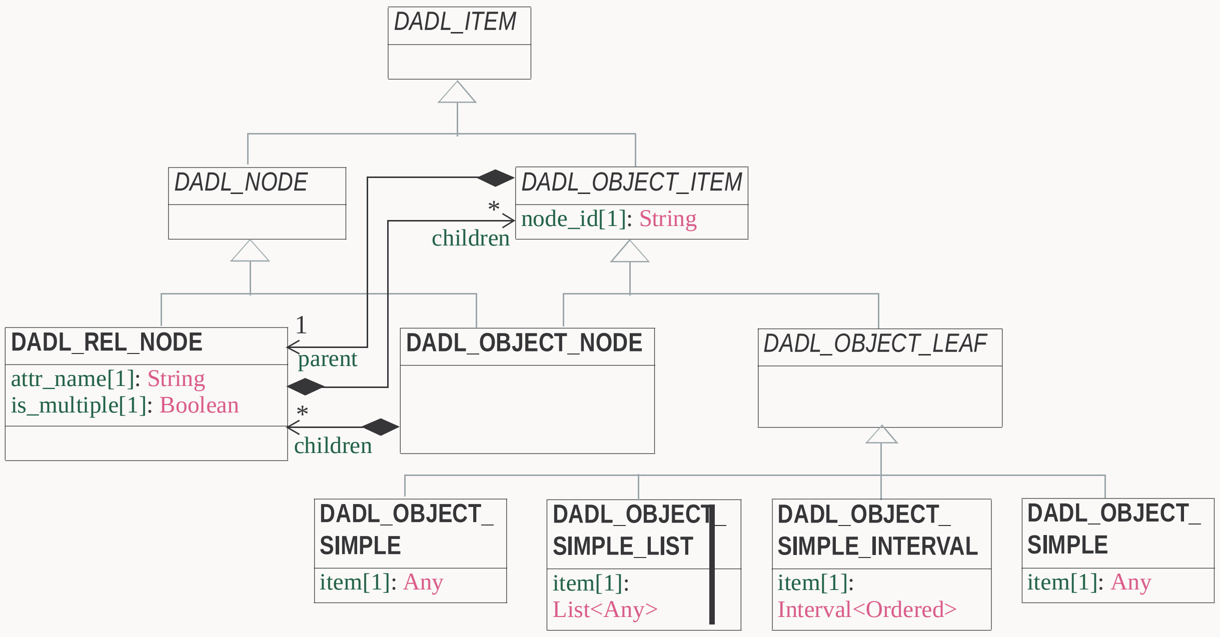


FIGURE 3 dADL Object Model

3 cADL - Constraint ADL

3.1 Overview

cADL is a syntax which enables constraints on data defined by object-oriented information models to be expressed in archetypes or other knowledge definition formalisms. It is most useful for defining the specific allowable constructions of data whose instances conform to very general object models. cADL is used both at “design time”, by authors and/or tools, and at runtime, by computational systems which validate data by comparing it to the appropriate sections of cADL in an archetype. The general appearance of cADL is illustrated by the following example:

```
PERSON[at0000] matches {                    -- constraint on PERSON instance
    name matches {                          -- constraint on PERSON.name
       TEXT matches {/.+/}                  -- any non-empty string
    }
    addresses cardinality matches {0..*} matches { -- constraint on
       ADDRESS matches {                    -- PERSON.addresses
          ...
       }
    }
    invariant:
       basic_validity: exists addresses implies exists name
}
```

Some of the textual keywords in this example can be more efficiently rendered using common mathematical logic symbols. In the following example, the `matches`, `exists` and `implies` keywords have been replaced by appropriate symbols:

```
PERSON[at0000] ∈ {                          -- constraint on PERSON instance
    name ∈ {                                -- constraint on PERSON.name
       TEXT ∈ {/..*/}                       -- any non-empty string
    }
    addresses cardinality ∈ {0..*} ∈ { -- constraint on
       ADDRESS ∈ {                          -- PERSON.addresses
          ...
       }
    }
    invariant:
       basic_validity: ∃ addresses ⊃ ∃ name
}
```

The full set of equivalences appears below. Raw cADL is stored in the text-based form, to remove any difficulties with representation of symbols, to avoid difficulties of authoring cADL text in basic text editors which do not supply symbols, and to aid reading in English. However, the symbolic form might be more widely used due to the use of tools, and formatting in HTML and other documentary formats, and may be more comfortable for non-English speakers and those with formal mathematical backgrounds. This document uses both conventions. The use of symbols or text is completely a matter of taste, and no meaning whatsoever is lost by completely ignoring one or other format according to one’s personal preference.

In the standard cADL documented in this section, literal leaf values (such as the regular expression `/..*/` in the above example) are always constraints on a set of ‘standard’ widely-accepted primitive types, as described in the dADL section. Other more sophisticated constraint syntax types are described in cADL - Constraint ADL on page 25.

3.2 Basics

3.2.1 Keywords

The following keywords are recognised in cADL:

- matches, ~matches, is_in, ~is_in
- occurrences, existence, cardinality
- unordered, unique
- use_node, use_archetype

In cADL invariants, the following further keywords can be used:

- exists, for_all,
- and, or, xor, not, implies, true, false

Symbol equivalents for some of the above are given in the following table.

Textual Rendering	Symbolic Rendering	Meaning
matches, is_in	∈	Set membership, "p is in P"
exists	∃	Existence quantifier, "there exists ..."
for_all	∀	Universal quantifier, "for all x..."
implies	⊃	Material implication, "p implies q", or "if p then q"
and	∧	Logical conjunction, "p and q"
or	∨	Logical disjunction, "p or q"
xor	⊻	Exclusive or, "only one of p or q"
not, ~	~	Negation, "not p"

The not operator can be applied as a prefix operator to all other operators except `for_all`; either textual rendering "not" or "~" can be used.

Keywords are shown in blue in this document.

The `matches` or `is_in` operator deserves special mention, since it is a key operator in cADL. This operator can be understood mathematically as set membership. When it occurs between a name and a block delimited by braces, the meaning is: the set of values allowed for the entity referred to by the name (either an object, or parts of an object - attributes) is specified between the braces. What appears between any matching pair of braces can be thought of as a *specification for a set of values*. Since blocks can be nested, this approach to specifying values can be understood in terms of nested sets, or in terms of a value space for objects of a set of defined types. Thus, in both of the following examples, the `matches` operator links the name of an entity to a value space.

```
XXX matches {/*ion/}  -- the set of english words ending in 'ing'

XXX matches {
    aaa matches {                        |
       YYY matches {0..3}                |
    }                                    | the value space of the
    bbb matches {                        | and instance of XXX
       ZZZ matches {>1992-12-01}         |
    }                                    |
}
```

Very occasionally, the `matches` operator needs to be used in the negative, usually at a leaf block. Any of the following will can be used to constraint the value space of XXX to any number except 5:

```
XXX ~matches {5}
XXX ~is_in {5}
XXX ∉{5}
```

The choice of whether to use matches or is_in is a matter of taste and background; those with a mathematical background will probably prefer `is_in`, while those with a data processing background may prefer `matches`.

3.2.2 Comments

In cADL, comments are indicated by the “--” characters. Multi-line comments are achieved using the “--” leader on each line where the comment continues. In this document, comments are shown in brown.

3.2.3 Information Model Identifiers

As with dADL, identifiers from the underlying information model are used. In cADL, type names and any property (i.e. attribute or function) name can be used, whereas in dADL, only type names and attribute names appear. Type identifiers are shown in this document in all uppercase, e.g. `PERSON`, while attribute identifiers are shown in all lowercase, e.g. `home_address`. In both cases, underscores are used to represent word breaks. This convention is solely for improving the readability of this document, and other conventions may be used, such as the common programmer’s mixed-case convention exemplified by `Person` and `homeAddress`. The convention chosen for any particular cADL document should be chosen based on the convention used in the underlying information model. Identifiers are shown in green in this document.

3.2.4 Node Identifiers

In cADL, an entity in brackets e.g. `[xxxx]` is used to identify “object nodes”, i.e. nodes expressing constraints on instances of some type. Object nodes always commence with a type name. Any string may appear within the brackets, depending on how it is used. Node identifiers are shown in magenta in this document.

3.2.5 Natural Language

cADL is completely independent of all natural languages. The only potential exception is where constraints include literal values from some language, and this is easily, and routinely avoided by the use of separate language and terminology definitions, as used in ADL archetypes. However, for the purposes of readability, comments in English have been included in this document to aid the reader. In real cADL documents, comments can be written in any language.

3.3 Structure

cADL constraints are written in a block-structured style, similar to block structured programming languages like C. A typical block resembles the following (the recurring pattern `/.+/` is a regular expression meaning “non-empty string”):

```
PERSON[001] matches {
    name ∈ {
       PERSON_NAME[002] ∈ {
          forenames cardinality ∈ {1..*} ∈ {/.+/}
          family_name ∈ {/.+/}
          title ∈ {“Dr”, “Miss”, “Mrs”, “Mr”, ...}
```

```
            }
         }
         addresses cardinality ∈ {1..*} ∈ {
            LOCATION_ADDRESS[003] ∈ {
               street_number existence ∈ {0..1} ∈ {/.+/}
               street_name ∈ {/.+/}
               locality ∈ {/.+/}
               post_code ∈ {/.+/}
               state ∈ {/.+/}
               country ∈ {/.+/}
            }
         }
     }
```

In the above, any identifier (shown in green) followed by the ∈ operator (equivalent text keyword: `matches` or `is_in`) followed by an open brace, is the start of a "block", which continues until the closing matching brace (normally visually indented to come under the start of the line at the beginning of the block).

The example above expresses a constraint on an instance of the type PERSON; the constraint is expressed by everything inside the PERSON block. The two blocks at the next level define constraints on properties of PERSON, in this case *name* and *addresses*. Each of these constraints is expressed in turn by the next level containing constraints on further types, and so on. The general structure is therefore a nesting of constraints on types, followed by constraints on properties (of that type), followed by types (being the types of the attribute under which it appears) and so on.

3.3.1 Information Model Entities

It may by now be clear that the identifiers in the above could correspond to entities in an object-oriented information model. A UML model compatible with the example above is shown in FIGURE 4.

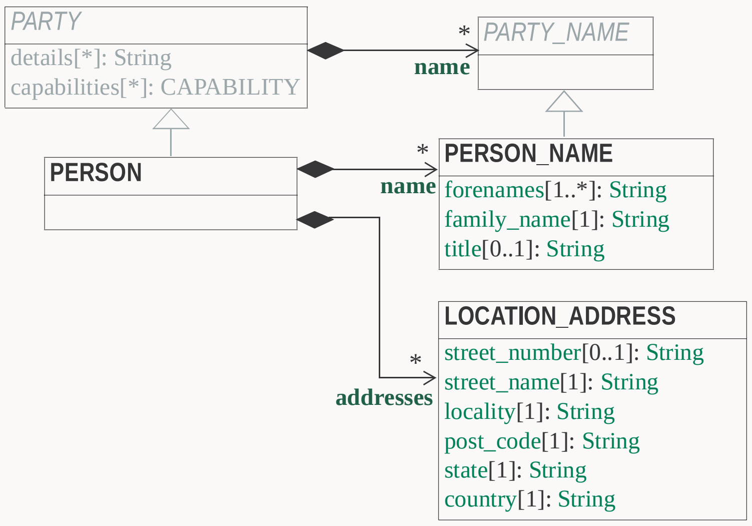


FIGURE 4 UML Model of PERSON

Note that there can easily be more than one model compatible with a given fragment of cADL syntax, and in particular, there may be more properties and classes in the reference model than are mentioned in the cADL constraints. In other words, a cADL text includes constraints *only for those parts of a model which are useful or meaningful to constrain*.

Constraints expressed in cADL cannot be stronger than those from the information model. For example, the PERSON.*family_name* attribute is mandatory in the model in FIGURE 4, so it is not valid to express a constraint allowing the attribute to be optional. In general, a cADL archetype can only further constrain an existing information model. However, it must be remembered that for very generic models consisting of only a few classes and a lot of optionality, this rule is not so much a limitation as a way of adding meaning to information. Thus, for a demographic information model which has only the types `PARTY` and `PERSON`, one can write cADL which defines the concepts of entities such as `COMPANY`, `EMPLOYEE`, `PROFESSIONAL`, and so on, in terms of constraints on the types available in the information model.

`To Be Continued:        moe generic PERSON ex required`

This general approach can be used to express constraints for instances of any information model. An example showing how to express a constraint on the *value* property of an `ELEMENT` class to be a `QUANTITY` with a suitable range for expressing blood pressure is as follows:

```
ELEMENT[10] matches {        -- diastolic blood pressure
    value matches {
       QUANTITY matches {
          magnitude matches {0..1000}
          property matches {"pressure"}
          units matches {"mm[Hg]"}
       }
    }
}
```

3.3.2 Existence, Cardinality and Occurrences

In any information model, there are *existence invariants* and *cardinalities* on *properties* (i.e. attributes and relationships). Existence invariants say whether an attribute must exist, and are indicated by “0..1” or “1” markers at line ends in UML diagrams (and usually mistakenly referred to as a “cardinality of 1..1”). Cardinalities indicate limits for the number of members of container types such as lists and sets. cADL makes the difference between these often confused characteristics clear. A further constraint that can occur in a cADL model is denoted by an *occurrences* constraint, which is used only on object nodes.

3.3.2.1 Existence

Existence constraints apply to properties only, and are expressed as follows:

```
QUANTITY matches {
    units existence matches {0..1} matches {“mm[Hg]”}
}
```

The meaning of an existence constraint is to indicate whether a value - i.e. an object - is required in runtime data for the property in question. The above example indicates that a value for the ‘units’ attribute is optional. The same logic applies whether the proeprty is of single or multiple cardinality, i.e. whether it is a container or not. For multiple properties, the existence constraint indicates whether the whole container (usually a list or set) is there; a further cardinality constraint (see below) indicates how many members in the container are allowed.

Existence is shown using the same constraint language as the rest of the archetype definition. Existence constraints can take the values `{0}`, `{0..0}`, `{0..1}`, `{1}`, or `{1..1}`. The first two of these constraints may not seem initially obvious, but may be reasonable in some cases: they say that an attribute must not be present in the particular situation modelled by the archetype. The default existence constraint, if none is shown, is `{1..1}`.

3.3.2.2 Cardinality and Built-in Container Types

While an existence constraint indicates the existence or not of a property value, a further constraint is needed to indicate the multiplicity of the property - in other words, to indicate that the type of the attribute is a container, e.g. List<T>. The *cardinality* constraint is used for this purpose.

Consider the following example:

```
HISTORY[001] occurrences ∈ {1} ∈ {
    periodic ∈ {False}
    events ∈ {
       EVENT[002] ∈ {}                                      -- 1 min sample
       EVENT[003] ∈ {}                                      -- 2 min sample
       EVENT[004] ∈ {}                                      -- 3 min sample
    }
}
```

This fragment says that the type `HISTORY` has a property *events*. Since no `existence` constraint is given, the default is `{1..1}`, meaning that *events* is a property of type `EVENT` in the reference model. The meaning of the next three lines is that the runtime data must follow the model of any one of the three constraints on `EVENT`. However, the intention of this fragment is most likely to be to constrain the data to be a `HISTORY` of multiple events, where the property *events* is actually of type `List<EVENT>`. To indicate this properly, a cardinality constraint is required on the *events* line, as in the following:

```
HISTORY[001] occurrences ∈ {1} ∈ {
    periodic ∈ {False}
    events cardinality ∈ {*} ∈ {
       EVENT[002] occurrences ∈ {0..1} ∈ {}                 -- 1 min sample
       EVENT[003] occurrences ∈ {0..1} ∈ {}                 -- 2 min sample
       EVENT[004] occurrences ∈ {0..1} ∈ {}                 -- 3 min sample
    }
}
```

The keyword `cardinality` indicates firstly that the property events must be of a container type, such as `List<T>`, `Set<T>`, `Bag<T>`, but does not indicate which - this is defined by the information model. The `{*}` constraint indicates that there may be 0 to many `EVENTs`, by default assumed to be in a List, i.e. an ordered, non-unique membership linear container. To specify a Set, the `unordered; unique` qualifiers would be used, as per the following examples:

```
    events cardinality ∈ {*; unordered} ∈ {
    events cardinality ∈ {*; unique} ∈ {
    events cardinality ∈ {*; unordered; unique} ∈ {
```

If cardinality is not mentioned, the property is assumed to be a non-container type, that is, a single relationship to which cardinality does not apply.

Cardinality and existence constraints can co-occur, in order to indicate various combinations on a container type property, e.g. that it is optional, but if present, is a container, as in the following:

```
    events existence ∈ {0..1} cardinality ∈ {0..*} ∈ {
```

3.3.2.3 Occurrences

A constraint on occurrences is used only with cADL object nodes, to indicate how many times in runtime data an instance of a given class conforming to a particular constraint can occur. In the example below, three `EVENT` constraints are shown; the first one (“1 minute sample”) is shown as mandatory, while the other two are optional.

```
    events cardinality ∈ {*} ∈ {
       EVENT[002] occurrences ∈ {1..1} ∈ {}                 -- 1 min sample
```

```
            EVENT[003] occurrences ∈ {0..1} ∈ {}                   -- 2 min sample
            EVENT[004] occurrences ∈ {0..1} ∈ {}                   -- 3 min sample
        }
```

Another contrived example below expresses a constraint on instances of GROUP such that for GROUPs representing tribes, clubs and families, there can only be one “head”, but there may be many members.

```
    GROUP[103] ∈ {
        kind ∈ {/tribe|family|club/}
        members cardinality ∈ {*} ∈ {
            PERSON[104] occurrences ∈ {1} matches {
                title ∈ {“head”}
                ...
            }
            PERSON[105] occurrences ∈ {0..*} matches {
                title ∈ {“member”}
                ...
            }
        }
    }
```

The first occurrences constraint indicates that a PERSON with the title “head” is mandatory in the GROUP, while the second indicates that at runtime, instances of PERSON with the title “member” can number from none to many. Occurrences may take the value of any range including {0..*}, meaning that any number of instances of the given class may appear in data, each conforming to the one constraint block in the archetype. A single positive integer, or the infinity indicator, may also be used on their own, thus: {2}, {*}. The default occurrences is none is mentioned is {1..1}.

3.3.3 Alternatives

Repeated blocks constraining objects of the same class can have two possible logical meanings, determined by the combination of the individual occurrences, and the cardinality of the containing property. Consider the following example:

```
    ELEMENT[04] matches {                         -- speed limit
        value matches {
            QUANTITY occurrences matches {0..1} matches {
                magnitude matches {0..60}
                property matches {"velocity"}
                units matches {"mi/h"}            -- miles per hour
            }
            QUANTITY occurrences matches {0..1} matches {
                magnitude matches {0..100}
                property matches {"velocity"}
                units matches {"km/h"}            -- km per hour
            }
        }
    }
```

Here, the cardinality of the *value* attribute is 1..1 (the default), while the occurrences of both QUANTITY constraints is optional, leading to the result that only one QUANTITY instance can appear in runtime data, and it can match either of the constraints. Conversely, the following example describes a HISTORY whose *events* property must contain one or more EVENTs, each of which can match any of the enclosed EVENT constraints.

```
    HISTORY[001] matches {
        events cardinality matches {1..*} matches {
            EVENT[002] occurrences matches {1..1} matches {...}
            EVENT[003] occurrences matches {0..*} matches {...}
            EVENT[004] occurrences matches {0..*} matches {...}
            ...
        }
    }
```

3.3.4 "Any" Constraints

There are two cases where it is useful to state a completely open, or "any", constraint. The "any" constraint is shown by a single asterisk (`*`) in braces. The first is when it is desired to show explicitly that some property can have any value, such as in the following:

```
    PERSON[001] matches {
        name existence matches {0..1} matches {*}
        ...
    }
```

The "any" constraint on *name* means that any value permitted by the underlying information model is also permitted by the archetype; however, it also provides an opportunity to specify an existence constraint which might be narrower than that in the information model. If the existence constraint is the same, an "any" constraint on a property is equivalent to no constraint being stated at all for that property in the cADL.

The second use of "any" as a constraint value is for types, such as in the following:

```
    ELEMENT[04] matches {                -- speed limit
        value matches {
           QUANTITY matches {*}
        }
    }
```

The meaning of this constraint is that in the data at runtime, the value property of `ELEMENT` must be of type `QUANTITY`, but can have any value internally. This is most useful for constraining objects to be of a certain type, without further constraining value, and is especially useful where the information model contains subtyping, and there is a need to restrict data to be of certain subtypes in certain contexts.

3.3.5 Node Identification

In many of the examples above, some of the object identifiers (i.e. the typenames) are followed by a node identifier, shown in brackets. Node identifiers are required for any node which is intended to be addressable elsewhere in the cADL text, or in the runtime system. Logically, all parent nodes of an identified node must also be identified back to the root node. The primary function of node identifiers is in forming paths, enabling cADL nodes to be unambiguously referred to. The node identifier can also perform a second function, that of giving a design-time *meaning* to the node, by equating the node identifier to some description. Thus, in the example above, the `ELEMENT` node is identified by the code `[10]`, which can be designated elsewhere in an archetype as meaning "diastolic blood pressure".

All nodes in a cADL text which correspond to nodes in data which might be referred to from elsewhere in the archetype, or might be used for querying at runtime, require a node identifier, and it is usually preferable to assign a design-time meaning, enabling paths and queries to be expressed using logical meanings rather than meaningless identifiers. When data is created according to a cADL specification, the node ids are written into the data, providing a reliable way of finding data nodes, regardless of what other runtime names might have been chosen by the user for the node in question. There are two reasons for this. Firstly, querying cannot rely on runtime names of nodes (e.g. names like "sys BP", "systolic bp", "sys blood press." entered by a doctor are unreliable for querying); secondly, it allows runtime data retrieved from a persistence mechanism to be re-associated with the cADL structure which was used to create it.

An example which clearly shows the difference between design-time meanings associated with node ids and runtime names is the following, for the root node of an SECTION headings hierarchy representing the problem/SOAP headings (a simple heading structure commonly used by clinicians to record patient contacts under top-level headings corresponding to the patient's problem(s), and under each problem heading, the headings "subjective", "objective", "assessment", and "plan").

```
SECTION[at0000] matches {                    -- problem
   name matches {
      CODED_TEXT matches {
         code matches {[ac0001]}
      }
   }
```

In the above, the node id [at0000] is assigned a meaning such as "clinical problem" in the archetype ontology section. The following lines express a constraint on the runtime *name* attribute, using the internal code [ac0001]. The constraint [ac0001] is also defined in the archetype ontology section with a formal statement meaning "any clinical problem type", which could clearly evaluate to thousands of possible values, such as "diabetes", "arthritis" and so on. As a result, in the runtime data, the node id corresponding to "clinical problem" and the actual problem type chosen at runtime by a user, e.g. "diabetes", can both be found. This enables querying to find all nodes which meaning "problem", or all nodes describing the problem "diabetes". Internal [ac] codes are described in Local Constraint Codes on page 45.

cADL on its own takes a minimalist approach to node identification. Node ids are required only where it is necessary to create paths, for example in invariants or "use" statements. However, the underlying reference model might have stronger requirements. The *open*EHR EHR information models [16] for example require that all nodes types which inherit from the class LOCATABLE have both a *meaning* and a runtime *name* attribute. Only data types (such as QUANTITY, CODED_TEXT) and their constituent types are exempt. The HL7 RIM [11] enforces a similar requirement: all archetype nodes which are based on Acts must have an *id* and usually a *code* (these two attributes are the equivalent of the *open*EHR *meaning* attribute, and the ADL node id); optionally they can also have a runtime name in the *code.original_text* attribute.

3.3.6 Paths

Paths are used in cADL to refer to cADL nodes. Recalling that the general hierarchical structure of cADL follows the pattern TYPE / property / TYPE / property /..., the syntax of paths takes exactly the same form. Paths are thus formed from an alternation of node identifiers and property names. The syntax used here is isomorphic to that used in the Xpath language, although the semantics of object hierarchies are slightly different. The syntax model of the main part of a path in cADL is the same as for dADL, i.e.:

```
['/'] attr_name ['[' object_id ']'] {'/' attr_name ['[' object_id ']' '/']}
```

However, property accessor names can be added at the end, separated by a dot ('.') in the usual object-oriented fashion, i.e.

```
object_path {['.' property_name ]}
```

This latter form is described further below.

Paths always refer to object nodes, and can only be constructed to nodes having node ids. The slash ('/') separator is used between path sections, and must always terminate a path. Lexically, a path can end either in a property name or a property name followed by a type node id in square brackets, which acts as an object identifier. The object identifier is required to differentiate between multiple children, but can be omitted, in which case the first child is assumed. This is only a sensible thing to do if there is known to be only one child, but is allowed, since it makes paths more readable in many cases. Unusually for a path syntax, object identifiers can be required, even if the property corresponds to a single relationship (as might be expected with the "name" property of an object) because in cADL, it is legal to supply multiple alternative object constraints for a relationship node which has single cardinality.

Paths are either *absolute*, i.e. they are assumed to start from the top of the cADL structure, or else *relative* to the node in which they are mentioned. Absolute paths always commence with an initial slash character, or with the id of the root node.

The following absolute paths are equivalent, and refer to a root node with the id `[001]`:

```
/
[001]/
```

The following absolute paths are equivalent, and refer to the object node at the "name" property of the root node:

```
/name/
[001]/name/
[001]/name[004]/
```

The following are examples of relative paths:

```
name/                -- the object node at the "name" property
period/              -- period property of an object node
items[003]/          -- the object node with id [003] at the "items" property
items[017]/          -- the object node with id [017] at the "items" property
```

It is valid to add object-oriented attribute references to the end of a path, using he dot ('.') character, if the underlying information model permits it, as in the following example.

```
/items/.count        -- count attribute of the items property
```

These examples are *physical* paths because they refer to object nodes using codes. Physical paths can be converted to *logical* paths using descriptive meanings for node identifiers, if defined. Thus, the following two paths might be equivalent:

```
[001]/members/
group/members/
```

The characters '.' and '/' must be quoted using the backslash ('\') character in logical path segments.

None of the paths shown here have any validity outside the cADL block in which they occur, since they do not include an identifier of the enclosing document, normally an archetype. To reference a cADL node in a document from elsewhere requires that the identifier of the document itself be prefixed to the path, as in the following archetype example:

```
[openehr-ehr-entry.apgar-result.v1]/
data[at0001]/event[at0002]/data[at0003]/
```

This kind of path expression is necessary to form the larger paths which occur when archetypes are composed to form larger structures.

3.3.7 Internal References

It is reasonably common that a particular inner block of cADL needs to be repeated later in the same outer block. Using a previously defined part of a cADL archetype is effected with the `use_node` keyword, in a line of the following form:

```
use_node TYPE object_path
```

This statement says: use the node of type `TYPE`, found at (the existing) path `object_path`. The following example shows the definitions of the `ADDRESS` nodes for phone, fax and email for a home `CONTACT` being reused for a work `CONTACT`.

```
PERSON[001] ∈ {
    identities ∈ {
        ...
    }
    contacts cardinality ∈ {0..*} ∈ {
        CONTACT[002] ∈ {                                -- home address
            purpose ∈ {...}
            addresses ∈ {...}
        }
        CONTACT[003] ∈ {                                -- postal address
            purpose ∈ {...}
            addresses ∈ {...}
        }
        CONTACT[004] ∈ {                                -- home contact
            purpose ∈ {...}
            addresses cardinality ∈ {0..*} ∈ {
                ADDRESS[005] ∈ {                        -- phone
                    type ∈ {...}
                    details ∈ {...}
                ADDRESS[006] ∈ {                        -- fax
                    type ∈ {...}
                    details ∈ {...}
                }
                ADDRESS[007] ∈ {                        -- email
                    type ∈ {...}
                    details ∈ {...}
                }
            }
        }
        CONTACT[008] ∈ {                                -- work contact
            purpose ∈ {...}
            addresses cardinality ∈ {0..*} ∈ {
                use_node ADDRESS [001]/contacts[004]/addresses[005]/ -- phone
                use_node ADDRESS [001]/contacts[004]/addresses[006]/ -- fax
                use_node ADDRESS [001]/contacts[004]/addresses[007]/ -- email
            }
        }
    }
```

3.3.8 Invariants

While most constraints are expressible using the cADL structured syntax, there are some which are more complex, and more easily expressed as *invariants*. Any constraint which relates more than one property to another is in this category, as are most constraints containing mathematical or logical formulae. An invariant statement is a first order predicate logic statement which can be evaluated to a boolean result at runtime. Objects and properties are referred to using paths. Invariant statements occur in sections at the end of type blocks, introduced by the `invariant` keyword, and are each preceded by a tag name, indicating the purpose of the invariant. The following simple example says that the speed in kilometres of some node is related to the speed-in-miles by a factor of 1.6:

```
invariant
```

```
        validity: [001]/speed[002]/kilometres/ = [003]/speed[004]/miles/ * 1.6
To Be Continued:        the '1.6' above should be coded and included in the ontol-
            ogy section
```

Invariant expressions can include the following operators:

universal quantifier: **for_all**

boolean operators: **not**, **and**, **or**, **xor**, **implies**, **exists**

relational operators: **=**, **<**, **>**, **<=**, **>=**, **!=**

arithmetic operators: +, -, *, /, ^, //, \\

set operators: **is_in** (i.e. member_of)

The textual operators among the above all have symbolic equivalents, described at the beginning of this chapter. Parentheses can be used to override standard precedence rules. Operands in an invariant expression can be any of the following:

manifest constant: any constant of any primitive type, expressed according to the dADL syntax for values

property reference: a path referring to a property, i.e. any path ending in ".property_name"

object reference: a path referring to an object node, i.e. any path ending in a node identifier

The only valid property reference operands (i.e. property and object nodepaths) which can appear in a given invariant are those referring to nodes and properties inside the block in which the invariant appears. The primary guideline for writing an invariant section is that it should be placed in the inner-most block possible, and refer to entities via relative path names. Invariants occur only in cADL object node blocks, and multiple invariant sections can occur in the one cADL archetype, giving the following general structure:

```
TYPE[xx] ∈ {
    xxxx ∈ {
       TYPE ∈ {
          xxx ∈ {xxxx}
          xxxx ∈ {xxxx}

          invariant
             tag_name: invariant expression
             tag_name: invariant expression
       }
       TYPE ∈ {
          xxx ∈ {xxxx}

          invariant
             tag_name: invariant expression
       }
    }
    invariant
       tag_name: invariant expression
       tag_name: invariant expression
       tag_name: invariant expression
}
```

The invariant part of cADL is in effect a small syntax of its own; it is close to a subset of the OMG's emerging OCL (Object Constraint Language) syntax and is very similar to the assertion syntax which has been used in the Object-Z [13] and Eiffel [9] languages and tools for over a decade. (See Kilov & Ross [7] and Sowa [10] for an explanation of predicate logic in information modelling.)

3.3.9 Archetype References

At any point in a cADL definition, other archetypes may be referred to, rather than defining the desired block inline. This might happen inside a PERSON archetype for example, where an ADDRESS archetype is referred to. The full semantics of archetype composition are described under Archetype Composition on page 52. References to archetypes are themselves constraints on the possible archetypes which are allowed at the point at which the reference occurs. Occasionally, they may refer to specific archetypes, but in general, the intention of archetypes is to provide general re-usable models of real world concepts; local constraints are left to templates. Accordingly, archetype references are expressed using cADL invariant syntax, with the keyword use_archetype prepended. The following example shows how the "Objective" SECTION in a problem/SOAP headings archetype refers to possible ENTRY and SECTION archetypes which are allowed under the *items* property.

```
SECTION[at2000] occurrences ∈ {0..1} ∈ {          -- objective
    items ∈ {
       use_archetype ENTRY occurrences ∈ {0..1} ∈ {
          identifier ∈ {/.*\.iso-ehr\.entry\..*\..*/}
       }
       use_archetype SECTION occurrences ∈ {0..1} ∈ {
          identifier ∈ {/.*\.iso-ehr\.section\..*\..*/}
       }
    }
}
```

Here, every constraint inside the block starting on a use_archetype line contains constraints not on the type referred to in the starting line (such as ENTRY), but on the archetype as a whole. Other constraints are possible as well, including that the allowed archetype must contain a certain keyword, or a certain path. The latter is quite powerful – it allows archetypes to be linked together on the basis of context. For example, under a "genetic relatives" heading in a Family History Organiser archetype, the following logical constraint might be used:

```
use_archetype ENTRY occurrences matches {0..*} matches {
    exists Family History Subject/subject/.relation
    and then Family History Subject/subject/.relation matches {
       CODED_TEXT matches {
          code matches {[ac0003]}     -- "parent" or "sibling"
       }
    }
}
```

To Be Determined: the exact syntax for the path and the constraint here needs to be worked out. The path cannot use local archetype ids since we do not know what archetype we are talking about.

3.4 Constraints on Primitive Types

While constraints on complex types follow the rules described so far, constraints on attributes of primitive types in cADL can be expressed without type names, and omitting one level of braces, as follows:

```
some_attr matches {some_pattern}
```

rather than:

```
some_attr matches {
    BASIC_TYPE matches {
       some_pattern
    }
}
```

Since all leaf attributes of all object models are of primitive types, or lists or sets of them, cADL archetypes using the brief form for primitive types are significantly less verbose overall, as well as being more directly comprehensible to human readers. cADL does not however oblige the brief form described here, and the more verbose one can be used. In either case, the syntax of the pattern appearing within the final pair of braces obeys the rules described below.

3.4.1 Constraints on String

Strings can be constrained in two ways: using a fixed string, and using a regular expression. An example of the first is:

```
species matches {“platypus”}
```

This forces the runtime value of the *species* attribute of some object to take the value “platypus”. **In almost all cases, this kind of string constraint should be avoided**, since it usually renders the body of the archetype language-dependent. Exceptions are proper names (e.g. “NHS”, “Apgar”), product tradenames (but note even these are typically different in different language locales, even if the different names are not literally translations of each other). The preferred way of constraing string attributes in a language independent way is with local [ac] codes. See Local Constraint Codes on page 45.

The second way of constraining strings is with regular expressions, a widely used syntax for expressing patterns for matching strings. The regular expression syntax used in cADL is a proper subset of that used in the Perl language (see [17] for a full specification of the regular expression language of Perl). Three uses of it are accepted in cADL:

```
string_attr matches {/regular expression/}
string_attr matches {=~ /regular expression/}
string_attr matches {!~ /regular expression/}
```

The first two are identical, indicating that the attribute value must match the supplied regular expression. The last indicates that the value must *not* match the expression. If the delimiter character is required in the pattern, it must be quoted with the backslash (‘\’) character, or else alternative delimiters can be used, enabling more comprehensible patterns. A typical example is regular expressions including units. The following two patterns are equivalent:

```
units matches {/km\/h|mi\/h/}
units matches {^km/h|mi/h^}
```

The rules for including special characters within strings follow those for dADL. In regular expressions, the small number of special characters are quoted according to the rules of Perl regular expressions; all other characters are quoted using the ISO and XML conventions described in the section on dADL.

The regular expression patterns supported in cADL are as follows.

Atomic Items

`.` match any single character. E.g. `/ ... /` matches any 3 characters which occur with a space before and after;

`[xyz]` match any of the characters in the set `xyz`. E.g. `/[0-9]/` matches any string containing a single decimal digit;

`[a-m]` match any of the characters in the set of characters formed by the continuous range from `a` to `m`. E.g. `/[0-9]/` matches any single character string containing a single decimal digit;

`[^xyz]` match any character except those in the set of characters formed by the continuous range from `a` to `m`. E.g. `/[^0-9]/` matches any single character string as long as it does not contain a single decimal digit;

Grouping

`(pattern)` parentheses are used to group items; any pattern appearing within parentheses treated as an atomic item for the purposes of the occurrences operators. E.g. `/([0-9][0-9])/` matches any 2-digit number.

Occurrences

`*` match 0 or more of the preceding atomic item. E.g. `/.*/` matches any string; `/[a-z]*/` matches any non-empty alphabetic string;

`+` match 1 or more occurrences of the preceding atomic item. E.g. `/a.+/` matches any string starting with 'a', followed by at least one further character;

`?` match 0 or 1 occurrences of the preceding atomic item. E.g. `/ab?/` matches the strings `"a"` and `"ab"`;

`{m,n}` match m to n occurrences of the preceding atomic item. E.g. `/ab{1,3}/` matches the strings `"ab"` and `"abb"` and `"abbb"`; `/[a-z]{1,3}/` matches all alphabetic strings of one to three characters in length;

`{m,}` match at least m occurrences of the preceding atomic item;

`{,n}` match at most n occurrences of the preceding atomic item;

`{m}` match exactly m occurrences of the preceding atomic item;

Special Character Classes

`\d`, `\D` match a decimal digit character; match a non-digit character;

`\s`, `\S` match a whitespace character; match a non-whitespace character;

Alternatives

`pattern1|pattern2` match either pattern1 or pattern2. E.g. `/lying|sitting|standing/` matches any of the words `"lying"`, `"sitting"` and `"standing"`.

A similar warning should be noted for the use of regular expressions to constraint strings: they should be limited to non-ligusitically dependent patterns, such as proper and scientific names. The use of regular expressions for constraints on normal words will render an archetype linguistically dependent, and potentially unusable by others.

3.4.2 Constraints on Integer

Integers can be constrainted with a single integer value, an integer interval, or a list of integers. For example:

```
length matches {1000}           -- limit to 100 exactly
length matches {950..1050}      -- allow +/- 50
length matches {0..1000}
```

The allowable syntax for integer values in ranges is as follows:

`infinity, -infinity, *` indicates infinity. '*' means plus or minus infinity depending on context;

`N` exactly this value, where N is an integer, or the infinity indicator;

`!= N` does not equal N;

Intervals can be expressed using the interval syntax from dADL, described in Intervals of Ordered Primitive Types on page 22. Intervals may be combined in integer constraints, using the semicolon character (';') as follows:

```
    normal_range matches {|10..100|}
    critical_range matches {|5..9; 101..110|}
```

Lists of integers expressed in the syntax from dADL, described in Lists of Primitive Types on page 22, can be used as a constraint, e.g.

```
    magnitude matches {0, 5, 8}
```

Note that a list used in this example indicates that the magnitude must match any one of the values in the list; if a cadinality indicator had been used, as in the following, the meaning would have been that magnitude is constrained to be the whole list of integers:

```
    magnitude cardinality matches {0..*} matches {0, 5, 8}
```

To Be Determined: In the future, functions may be allowed e.g. `{f(x):(x\\4=0)}` means "x ok if divisible by 4"

To Be Determined: An alternative syntax for interval values is as used by HL7, e.g. [lower;upper]. This might be a better syntax. Qs: what is more readable, how are disjoint intervals & exclusion specified in this syntax?

3.4.3 Constraints on Real

Constraints on Real follow exactly the same syntax as for Integers, except that all real numbers are indicated by the use of the decimal point and at least one succeeding digit, which may be 0. Typical examples are:

```
    magnitude matches {5.5}                      -- fixed value
    magnitude matches {|5.5..6.0|}               -- interval
    magnitude matches {5.5, 6.0, 6.5}            -- list
```

3.4.4 Constraints on Boolean

Boolean runtime values can be constrained to be True, False, or either, as follows:

```
    some_flag matches {True}
    some_flag matches {False}
    some_flag matches {True, False}
```

3.4.5 Constraints on Character

Characters can be constrained in cADL using manifest values enclosed in single quotes, or using single-character regular expression elements, also enclosed in single quotes, as per the following examples:

```
    color_name matches {'r'}
    color_name matches {'[rgbcmyk]'}
```

The only allowed elements of the regular expression syntax in character expressions are the following:

- any item from the Atomic Items list above;
- any item from the Special Character Classes list above;
- the '.' character, standing for "any character";
- an alternative expression whose parts are any item types, e.g. `'a'|'b'|[m-z]`

3.4.6 Constraints on Dates, Times and Durations

Patterns

Dates, times, and date/times (i.e. timestamps), can be constrained in two ways. The first one, usually used in archetypes uses *patterns* based on the ISO 8601 date/time syntax, which indicate which parts of the date or time must be supplied. The following table shows the valid patterns which can be used, and the types implied by each pattern. The patterns are formed from the abstract pattern `yyyy-mm-dd hh:mm:ss` (itself formed by translating each field of an ISO 8601 date/time into a letter representing its type), with either `'?'` (meaning optional) or `'X'` (not allowed) characters substituted in appropriate places.

Implied Type	Pattern	Explanation
Date	`yyyy-mm-dd`	full date must be specified
Date, Partial Date	`yyyy-mm-??`	optional day; e.g. day in month forgotten
Date, Partial Date	`yyyy-??-??`	optional month, day; i.e. any date allowed
Partial Date	`yyyy-??-XX`	optional month, no day; (any examples?)
Time	`hh:mm:ss`	full time must be specified
Partial Time	`hh:mm:XX`	no seconds; e.g. appointment time
Partial Time	`hh:??:XX`	optional minutes, no seconds; e.g. normal clock times
Time, Partial Time	`hh:??:??`	optional minutes, seconds; i.e. any time allowed
Date/Time	`yyyy-mm-dd hh:mm:ss`	full date/time must be specified
Date/Time, Partial Date/Time	`yyyy-mm-dd hh:mm:??`	optional seconds; e.g. appointment date/time
Partial Date/Time	`yyyy-mm-dd hh:mm:XX`	no seconds; e.g. appointment date/time
Partial Date/Time	`yyyy-mm-dd hh:??:XX`	no seconds, minutes optional; e.g. in patient-recollected date/times
Date/Time, Partial Date/Time Partial Date/Partial Time	`yyyy-??-?? ??:??:??`	minimum valid date/time constraint

Literals

The second way of constraining dates and times, which will most likely occur only in local templates, is using actual values and ranges, in the same way as for Integers and Reals. In this case, the limit values are specified using the same patterns from the above table, but with numbers in the positions where `'X'` and `'?'` do not appear. For example, the pattern `yyyy-??-XX` could be transformed into `1995-??-XX` to mean any partial date in 1995. Constraints are then expressed according to the following examples:

```
    1995-??-XX                  -- any partial date in 1995
    |< 09:30:00|                -- any time before 9:30 am
    |<= 09:30:00|               -- any time at or before 9:30 am
    |> 09:30:00|               -- any time after 9:30 am
    |>= 09:30:00|              -- any time at or after 9:30 am
    2004-05-20..2004-06-02     -- a date range
```

Duration Constraints

Durations are constrained using absolute ISO 8601 values, or ranges of the same, e.g.:

```
    P0d0h1m0s            -- 1 minute
    P1d8h0m0s            -- 1 day 8 hrs
    |P0S..P1m30s|        -- Reasonable time offset of first apgar sample
```

3.4.7 Constraints on Lists of Primitive types

In many cases, the type in the information model of an attribute to be constrained is a list or set of primitive types. This must be indicated in cADL using the `cardinality` keyword (as for complex types), as follows:

```
    some_attr cardinality matches {0..*} matches {some_pattern}
```

The pattern to match in the final braces will then have the meaning of a list or set of value constraints, rather than a single value constraint.

To Be Determined: how to define such lists? How to indicate which values in teh data correspeond to which values in teh list; and if lists are open or closed?

3.5 cADL Object Model

FIGURE 5 illustrates the essentials of the cADL object model.

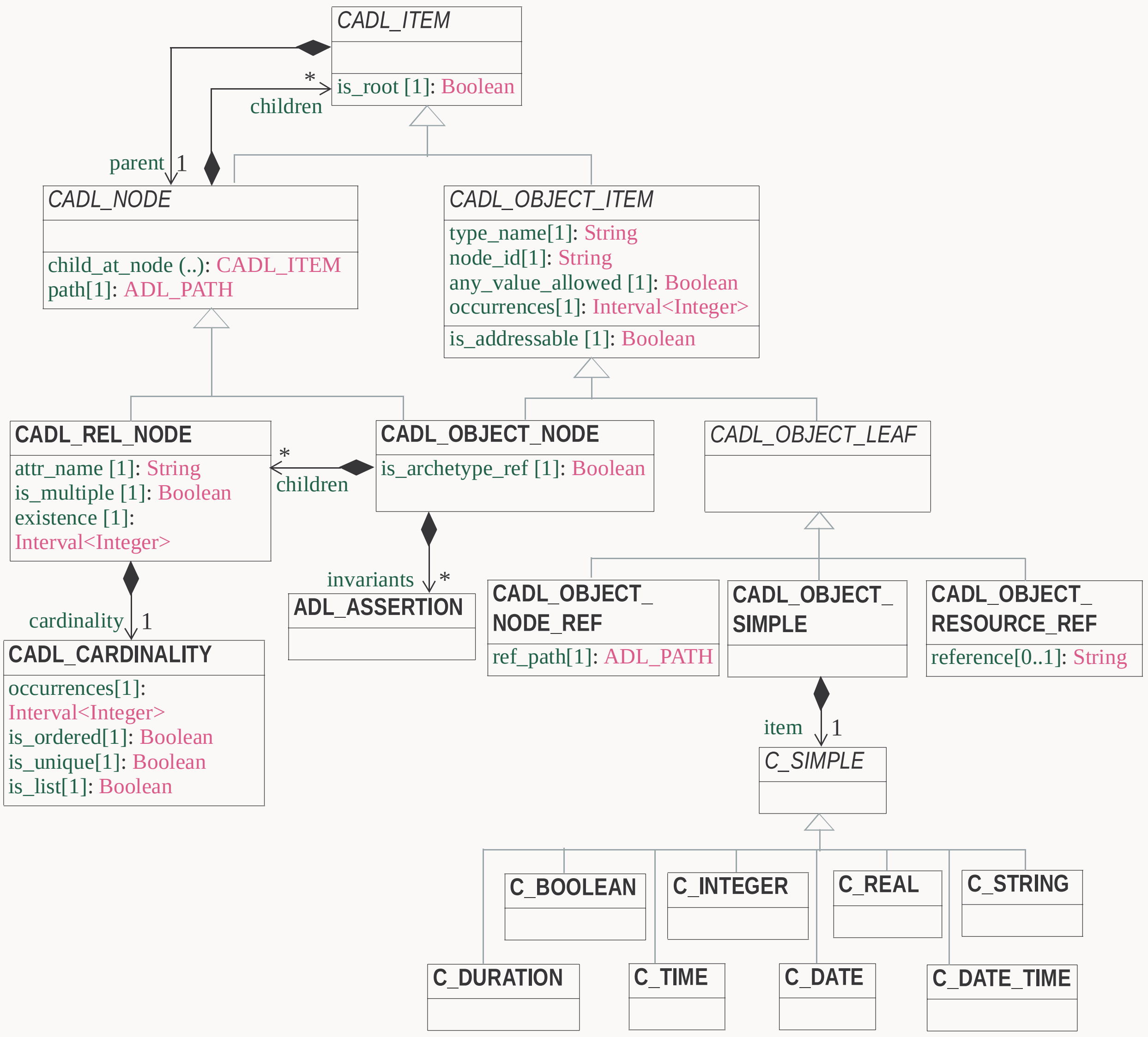


FIGURE 5 cADL Object Model

4 ADL - Archetype Definition Language

This section describes ADL archetypes as a whole, adding a small amount of detail to the descriptions of dADL and cADL already given. The important topic of the relationship of the cADL-encoded `definition` section and the dADL-encoded `ontology` section is discussed in detail. In this section, only standard ADL (i.e. the cADL and dADL constructs and types described so far) is assumed. Archetypes for use in particular domains can also be built with more efficient syntax and domain-specific types, as described in Predefined Type Libraries on page 59, and the succeeding sections.

An ADL archetype follows the structure shown below:

```
archetype
    archetype_id
specialize
    parent_archetype_id
concept
    coded_concept_name
description
    dADL meta-data section
definition
    cADL structural section
ontology
    dADL definitions section
```

4.1 Basics

4.1.1 Keywords

ADL has a small number of keywords which are reserved for use in archetype declarations, as follows:

- `archetype`, `specialise/specialize`, `concept`,
- `description`, `definition`, `ontology`

All of these words can safely appear as identifiers in the `definition` and `ontology` sections.

4.1.2 Node Identification

In the `definition` section of an ADL archetype, a particular scheme of codes is used for node identifiers as well as for denoting constraints on textual (i.e. language dependent) items. Codes are either local to the archetype, or from an external lexicon. This means that the archetype description is the same in all languages, and is available in any language that the codes have been translated to. All term codes are shown in brackets (`[]`). Codes used as node identifiers and defined within the same archetype are prefixed with "at" and have only 4 digits, e.g. `[at0010]`. Specialisations of locally coded concepts have the same root, followed by "dot" extensions, e.g. `[at0010.2]`. From a terminology point of view, these codes have no implied semantics - the "dot" structuring is used as an optimisation on node identification.

4.1.3 Local Constraint Codes

A second kind of local code is used to stand for constraints on textual items in the body of the archetype. Although these could be included in the main archetype body, because they are language- and/or terminology-sensitive, they are defined in the ontology section, and referenced by codes prefixed by "ac", e.g. `[ac0009]`. The use of these codes is described in section 4.4.4

4.2 Header Sections

4.2.1 Archetype Section

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

```
archetype
    mayo.openehr-ehr-entry.haematology.v1
```

The multi-axial identifier identifies archetypes in a global space. The syntax of the identifier is described under Archetype Identification on page 17 in The openEHR Archetype System.

4.2.2 Specialise Section

This optional section indicates that the archetype is a specialisation of some other archetype, whose identity must be given. Only one specialisation parent is allowed. An example of declaring specialisation is as follows:

```
archetype
    mayo.openehr-ehr-entry.haematology-cbc.v1
specialise
    mayo.openehr-ehr-entry.haematology.v1
```

Here the identifier of the new archetype is derived from that of the parent by adding a new section to its domain concept section. See Archetype Identification on page 17 in The openEHR Archetype System.

Note that both the US and British english versions of the word “specialise” are valid in ADL.

4.2.3 Concept Section

All archetypes represent some real world concept, such as a “patient”, a “blood pressure”, or an “antenatal examination”. The concept is always coded, ensuring that it can be displayed in any language the archetype has been translated to. A typical `concept` section is as follows:

```
concept
    [at0010]    -- haematology result
```

In this concept definition, the term definition of `[at0010]` is the proper description corresponding to the “`haematology-cbc`” section of the archetype id above.

4.2.4 Description Section

The `description` section of an archetype contains descriptive information, or what some people think of as document “meta-data”, i.e. items that can be used in repository indexes and for searching. The dADL syntax is used for the description, as in the following example:

```
description
    author = <"Sam Heard <s.heard@littlerock.con>">
    submission = <
       organisation = <"openEHR Foundation">
       date = <2003-06-10>
    >
    revision = <
       identifier = <"1.2">
       author = <"Thomas Beale <t.beale@home.net>">
       date = <2003-12-11>
    >
    status = <"draft">
```

```
        purpose = <
           items("en") = <"general model of haematology lab results">
           items("de") = <"Allgemeines Modell fuer Haematologie Laborwerte">
        >
        use = <
           items("en") = <"can be used directly or specialised for\
                         particular result types">
           items("de") = <"kann entweder direkt benutzt werden, oder speziell \
                         fuer besondere Ergebnissarten">
        >
        misuse = <>
```

A number of details are worth noting here. Firstly, the free hierarchical structuring capability of dADL is exploited for expressing the "deep" structure of the submission, revision, purpose and use items. Secondly, the dADL qualified list form is used to allow multiple translations of the purpose and use to be shown. Lastly, empty items such as misuse (structured if there is data) are shown with just one level of empty brackets. The above example shows meta-data based on the HL7 Templates Proposal [12] and the meta-data of the SynEx and GeHR archetypes.

Which descriptive items are required will depend on the semantic standards imposed on archetypes by health standards organisations and/or the design of archetype repositories and is not specified by ADL.

4.3 Definition Section

The definition section contains the main formal definition of the archetype, and is written in the Constraint Definition Language (cADL). A typical definition section is as follows:

```
    definition
        ENTRY[at0000] ∈ {           -- blood pressure measurement
           name ∈ {                 -- any synonym of BP
              CODED_TEXT ∈ {
                 code ∈ {
                    CODE_PHRASE ∈ {[ac0001]}
                 }
              }
           }
           data ∈ {
              HISTORY[at9001] ∈ {                         -- history
                 events cardinality ∈ {1..*} ∈ {
                    EVENT[at9002] occurrences ∈ {0..1} ∈ {-- baseline
                       name ∈ {
                          CODED_TEXT ∈ {
                             code ∈ {
                                CODE_PHRASE ∈ {[ac0002]}
                             }
                          }
                       }
                       data ∈ {
                          LIST_S[at1000] ∈ {      -- systemic arterial BP
                             items cardinality ∈ {2..*} ∈ {
                                ELEMENT[at1100] ∈ {             -- systolic BP
                                   name ∈ {       -- any synonym of 'systolic'
                                      CODED_TEXT ∈ {
                                         code ∈ {
                                            CODE_PHRASE ∈ {[ac0002]}
                                         }
```

```
            }
         }
         value ∈ {
            QUANTITY ∈ {
               magnitude ∈ {0..1000}
               property ∈ {[properties::0944]}
                                       -- “pressure”
               units ∈ {[units::387]}  -- “mm[Hg]”
            }
         }
      }
      ELEMENT[at1200] ∈ {          -- diastolic BP
         name ∈ {          -- any synonym of 'diastolic'
            CODED_TEXT ∈ {
               code ∈ {
                  CODE_PHRASE ∈ {[ac0003]}
               }
            }
         }
         value ∈ {
            QUANTITY ∈ {
               magnitude ∈ {0..1000}
               property ∈ {[properties::0944]}
                                       -- “pressure”
               units ∈ {[units::387]}  -- “mm[Hg]”
            }
         }
      }
      ELEMENT[at9000] occurrences ∈ {0..*} ∈ {*}
                                 -- unknown new item
   }
...
```

This definition expresses constraints on instances of the types ENTRY, HISTORY, EVENT, LIST_S, ELEMENT, QUANTITY, and CODED_TEXT so as to allow them to represent a blood pressure measurement, consisting of a history of measurement events, each consisting of at least systolic and diastolic pressures, as well as any number of other items (expressed by the [at9000] “any” node near the bottom).

4.4 Ontology Section

4.4.1 Overview

The ontology section of an archetype is expressed in dADL, and is where codes representing node meanings and constraints on text or terms, bindings to terminologies, other ontological definitions (such as quantitative definitions), and language translations are added. The following example shows the general layout of this section.

```
ontology
    primary_language = <“en”>
    languages_available = <“en”, “de”>
    terminologies_available = <“snomed_ct”, ...>

    term_definitions(“en”) = <
       ...
    >

    term_definitions(“de”)= <
```

```
            description = <translation = <"acme_translation@acme.com.de">
            ...
         >

         term_binding("snomed_ct") = <
            ...
         >

         constraint_definitions("en") = <
            ...
         >

         constraint_binding("snomed_ct") = <
            ...
         >
```

The `term_definitions` section is mandatory, and must be defined for each translation carried out.

Each of these sections can have its own meta-data, which appears within description sub-sections, such as the one shown above providing translation details.

4.4.2 Ontology Header Statements

The first three headings in the `ontology` section describe the primary language in which the archetype was authored (essential for evaluating natural language quality), the total languages available in the archetype, and the terminology bindings available. There can be only one primary language. The languages available list should be updated every time a translation of a `term_definition` and `constraint_definition` section is added, and must include the primary language as well. The terminologies_available statement includes the identifiers of all terminologies for which `term_binding` sections have been written.

4.4.3 Term_definition Section

This section is where all archetype local terms (that is, terms of the form `[atNNNN]`) are defined. The following example shows an extract from the english and german term definitions for the archetype local terms in a problem/SOAP headings archetype. Each term is defined using tagged values. ADL does not currently force any particular tags - validation of term definitions can be left until after the parsing process. However, it is expected that almost all term and constraint definitions will include at least the tags “text” and “description”, which are akin to the usual rubric, and full definition found in terminologies like SNOMED-CT.

```
     term_definitions("en") = <
         items("at0001") = <
            text = <"problem/SOAP headings">
            description = <"SOAP heading structure for multiple problems">
         >
         items("at0000") = <
            text = <"problem">
            description = <"The problem experienced by the subject of care to \
                                which the contained information relates">
         >
         ...
         items("at4000") = <
            text = <"plan">
            description = <"The clinician's professional advice">
         >
     >
```

```
    term_definitions(“de”) = <
        description = <
           translation = <
              provenance = <"mdarlison@chimera.upstairs.uk">
              quality_control = <"British Medical Translator’s id 00400595”>
           >
        >
        items(“at0001”) = <
           text = <"Problem/SOAP Schema">
           description = <"SOAP-Schlagwort-Gruppierungsschema fuer mehrfache \
                          Probleme">
        >
        items(“at0000”) = <
           text = <"klinisches Problem">
           description = <"Das Problem des Patienten worauf sich diese \
                          Informationen beziehen">
        >
        ...
        items(“at4000”) = <
           text = <"Plan">
           description = <"Klinisch-professionelle Beratung des Pflegenden">
        >
    >
```

In some cases, term definitions may have been lifted from existing terminologies (only a safe thing to do if the definitions *exactly* match the need in the archetype). To indicate where definitions come from, a “provenance” tag can be used, as follows:

```
    items(“at4000”) = <
        text = <"plan">;
        description = <"The clinician's professional advice">;
        provenance = <"ACME_terminology(v3.9a)">
    >
```

Note that this does not indicate a *binding* to any term; bindings are described in the binding sections.

4.4.4 Constraint_definition Section

The `constraint_definition` section is of exactly the same form as the `term_definition` section, and provides the definitions - i.e. the meanings - of the local constraint codes, which are of the form `[acNNNN]`. Each such code refers to some constraint such as “any term which is a subtype of ‘hepatitis’ in the ICD9AM terminology”; the constraint definitions do not provide the constraints themselves, but define the *meanings* of such constraints, in a manner comprehensible to human beings, and usable in GUI applications. This may seem a superfluous thing to do, but in fact it is quite important. Firstly, term constraints can only be expressed with respect to particular terminologies - a constraint for “kind of hepatitis” would be expressed in different ways for each terminology which the archetype is bound to. For this reason, the actual constraints are defined in the `constraint_binding` section. An example of a constraint term definition for the hepatitis constraint is as follows:

```
    items(“at1015”) = <
        text = <"type of hepatitis">
        description = <"any term which means a kind of viral hepatitis">
    >
```

Note that while it often seems tempting to use classification codes, e.g. from the ICD vocabularies, these will rarely be much use in terminology or constraint definitions, because it is nearly always *descriptive*, not classificatory terms which are needed.

4.4.5 Term_binding Section

This section is used to describe the equivalences between archetype local terms and terms found in external terminologies. The purpose is solely for allowing query engine software which wants to search for an instance of some external term to determine what equivalent to use in the archetype. Note that this is distinct from the process of embedding mapped terms in runtime data, which is also possible with the data models of HL7v3, openEHR, and CEN 13606.

A typical term binding section resembles the following:

```
term_binding("umls") = <
    items("at0000") = <[umls::C124305]> -- apgar result
    items("at0002") = <[umls::0000000]> -- 1-minute event
    items("at0004") = <[umls::C234305]> -- cardiac score
    items("at0005") = <[umls::C232405]> -- respiratory score
    items("at0006") = <[umls::C254305]> -- muscle tone score
    items("at0007") = <[umls::C987305]> -- reflex response score
    items("at0008") = <[umls::C189305]> -- color score
    items("at0009") = <[umls::C187305]> -- apgar score
    items("at0010") = <[umls::C325305]> -- 2-minute apgar
    items("at0011") = <[umls::C725354]> -- 5-minute apgar
    items("at0012") = <[umls::C224305]> -- 10-minute apgar
>
```

Each entry simply indicates which term in an external terminology is equivalent to the archetype internal codes. Note that not all internal codes necessarily have equivalents: for this reason, a terminology binding is assumed to be valid even if it does not contain all of the internal codes.

To Be Determined: future possibility: more than one binding to the same terminology for different purposes, or by different authors?

To Be Determined: need to handle numerous small domains defined by one authority e.g. HL7.

4.4.6 Constraint_binding Section

The last of the ontology sections formally describes text constraints from the main archetype body. They are described separately because they are terminology dependent, and because there may be more than one for a given logical constraint. A typical example follows:

```
constraint_binding("snomed_ct") = < -- is-a problem type
    items("ac0001") = <query("terminology",
                      "terminology_id = 'snomed_ct' AND
                      has_relation [102002] with_target [128004]")>
    items("ac0002") = <query("terminology",  -- subjective
                      "terminology_id = 'snomed_ct' AND
                      synonym_of [128025]")>
    items("ac0003") = <query("terminology", -- objective
                      "terminology_id = 'snomed_ct' AND
                      synonym_of [128009]")>
>
```

In this example, each local constraint code is formally defined to refer to the result of a query to a service, in this case, a terminology service which can interrogate the Snomed-CT terminology.

To Be Determined: Note that the query syntax used above is only for illustration; syntaxes for use with services like OMG HDTF TQS and HL7 CTS are being developed.

5 Archetype Relationships

This section describes the creation of new archetypes based on existing ones. There are two ways this can occur: due to versioning and due to specialisation. Revisions are included here, although they do not create a new archetype. As soon as we speak of related archetypes, we must also understand the relationship of the data. Accordingly, a notion of *archetype conformance* must be defined, as follows:

an archetype B conforms to an archetype A if all data created according to B are guaranteed to conform also to A

Specialisation creates conformant archetypes, while versioning does not.

5.1 New Versions

New versions of an existing archetype can be created. A new version is required for any change to an archetype which fixes an error, and creates as a result a non-conforming archetype. A "non-conforming" archetype is one whose data does not conform to the previous archetype. A data conversion algorithm must always be supplied with a new version. Versions are indicated in the archetype identifier, meaning that two versions of the same logical archetype are technically speaking two different archetypes.

To Be Determined: version number series - see earlier in archetype identifier section

5.2 New Revisions

Archetypes can also be revised, meaning that changes which do not compromise data created according to the earlier revision can be incorporated without creating a new version or a specialisation. Situations where revisions are used include:

- the addition of a new foreign language translation
- the addition of a new terminology binding
- weakening of some constraints, e.g. cardinalities being changed from 1..1 to 0..1
- changes to items in the description

Since a revised archetype is 100% backwards compatible with its predecessor, revision does not cause the archetype id to be changed. Revisions are indicated in the meta-data, and are numbered according to the series {1.0, 1.1, 1.2, ... 2.0 ....}.

5.3 Archetype Composition

Archetypes are designed to be composed into larger structures which describe whole sections of data in a system, for example whole documents in an EHR system, or an entire `PERSON` object in a demographic service. The `use_archetype` keyword introduces a constraint which evaluates to a set of possible archetypes which can be attached at the point where it appears. At runtime, the user has to choose one of these. Archetype references thus define the *possible* archetype compositions at runtime, although in some cases, they may mention a single specific archetype, meaning that no further choice is required at runtime. Generally archetypes should allow the widest possible choice of archetypes at each next level, with templates being used to define particular compositions (or "chaining") of archetypes. Typical compositions of archetypes based on the CEN 13606 EHR standard information model are `COMPOSITION` / `SECTION` / ... / `SECTION` / `ENTRY`.

5.3.1 Paths in Archetype Compositions

When archetypes are composed, paths can be used to refer to an item in a lower archetype, starting from the topmost archetype. For example, the combination of and SECTION archetype for “birth” and an ENTRY archetype for “apgar result”, will result in a set of paths which can be used to refer to all items in the apgar result, starting from the top organiser. The following path provides a typical example of such a path, in this case, referring to the cardiac score of an apgar result within a birth recording organiser structure.

```
/[openehr.openehr-ehr-organiser.birth.v1]/items[birth]/items[apgar]
    /[openehr.openehr-ehr-entry.apgar.v1]/data[history]/events
    [1min sample]/data[apgar list]/items[cardiac score]
```

All composite paths must be formed using external paths, i.e. paths containing the archetype identifier as the first section.

5.4 Specialisation

Archetypes can be specialised. The primary rule for specialisation is that *data created according to the more specialised archetype are guaranteed to conform to the parent archetype*. Specialised archetypes have an identifier based on the parent archetype id, but with a modified section, as described earlier.

Since ADL archetypes are designed to be usable in a standalone fashion, they include all the text of their definition. This applies to specialised archetypes as well, meaning that the contents of the ADL file of a specialised archetype contains all the relevant parts from its parent, with additions or modifications according to the specialisation rules. In an analogy with object-oriented class definitions, ADL archetypes can be thought of as always being in “inheritance-flattened” form. Validation of a specialised archetype requires that its parent be present, and relies on being able to locate equivalent sections using node identifiers. For this reason, nodes in specialised archetypes carry either the same identifiers as the corresponding nodes in the parent, or else a node identifier derived by “specialising” the parent node id, using “dot” notation. The following describes in detail how archetypes are specialised.

5.4.1 Object Nodes

An object node may be specialised by:

- reducing the `occurrences` constraint to any interval fitting inside the `occurrences` interval of the corresponding node in the parent;
- splitting the node into multiple nodes, each of a “subtype” of the parent node. Each child node must contain constraints which inside the set of the corresponding parent nodes. Subtyped nodes are indicated with node ids which are specialisations of the parent node id, and have specialised meanings. For example, a parent node id of `[at0203]` may have specialisations formed by adding a dot and further digits, e.g. `[at0203.1]`, `[at0203.2]` etc. The terminological meanings of the new codes should be subtypes of that of the original parent node.
- “use” nodes which referred to the original node in the parent archetype may be left intact, meaning that any of the new nodes may be used, or may be redefined to use a smaller subset of the child nodes.

5.4.2 Relationship Nodes

A relationship node may be specialised by:

- narrowing its existence or cardinality constraint to one fitting inside the corresponding constraints of the parent relationship.

5.4.3 Leaf Nodes

A leaf object node may be specialised by:

- narrowing the valid values allowed by its constraints, for example, reducing integer intervals; reducing the set of allowed string patterns and so on.

5.4.4 Text constraint target nodes

Text constraint target nodes defined in the constraint bindings section of the archetype may only be narrowed to evaluate to a subset of the terms resulting from evaluating the parent constraint. The actual set of terms resulting from evaluating such a constraint may change, due to evolution of terminology (the set of subtypes of hepatitis has expanded in recent years, for example), but the set *definitions* must remain the same. If an original constraint indicated only terms from ICD10, for example, a specialised archetype could not allow SNOMED terms in the same place, since they would not be valid according to the parent constraint.

5.4.5 Archetype Term Definitions

Local term definitions defined in the terminology part of the archetype can only be added to, with terms which are specialised versions of the existing terms.

6 Relationship with Language and Ontology

6.1 The Problem of Terminology

ADL archetypes take a particular stance on interfacing with the terminology world. The basic tenets of the relationship between archetypes (whether or not expressed in ADL) and ontological and terminological resources relate to the linguistic elements in an archetype definition, which are as follows:

- the names of nodes, e.g. “severity”;
- the allowed values of textual nodes, e.g. “low”, “mild”, “medium”, “high”;
- codes, usually from classification, attached to name/value nodes.

The last of these is easy to achieve, and it is the first two that concern us here. The tenets are as follows:

1. every linguistic element in an archetype must have a *defined meaning*, available in each language for which a translation has been done.

The consequence of this is that each such element must be coded within the archetype. The naïve approach is to think that coded terms from external terminologies can be used directly for this purpose. A number of factors make this impossible (in the following “terminology” refers to descriptive terminologies like Snomed-CT, or ontologies, like Galen, not classifications like ICD10):

2. there may be no terminology containing the required term;
3. there may be one or more terminologies containing what appears lexically to be the required term, but there is no guarantee that any of these really does encode the intended meaning; in any case, there would be no way to resolve multiple conpeting definitions;
4. even if the term is found in a terminology, there might not be the required language in the its translations; since most external terminologies are large, there is no easy way to effect a translation, apart from an unofficial local translation of one or two terms.

The underlying reason for these problems is that existing terminologies and ontologies are largely contextless - they are intended for many uses - while archetypes are focussed ontological “capsules” of meaning - the words and terms used in them relate specifically to the subject of the archetype. Thus, in general, the correspondence between the required names and meanings in a given archetype, and the (roughly) matching terms in an external terminology will always be approximate, and may even be misleading or wrong. The only conclusion that can be drawn from this state of affairs is:

5. all names and textual value constraints in archetypes have to be locally defined within the archetype.

This has a number of fortuitous side-effects:

- archetypes can be written without the use of terminologies (this is not necessarily a recommendation, of course);
- translations can occur on a per-archetype basis, making the job of translation much smaller (it may only be 20 terms) and of higher-quality (the exact meanings of the terms and value constraints is clear within the archetype, whereas translators of whole terminologies always face the problem of multiple possible meanings of each word);

6.2 The Need for Shared Meaning

At first sight, while local coding in archetypes makes archetypes easier and more correct, it appears to forego the supposed most important advantage of shared terminology, which is shared meaning. The assumption has always been that the very fact of common use of a large terminology means that human users and computer systems can understand any datum the same way if it is coded with the same code. Common use of terms in data by software is the key to any kind of reliable inferencing in decision support, so these requirements are not unimportant. There are two responses to this. Firstly, in many cases the supposed sharability might not exist, given the problem of multiple meanings of terms and words, depending on context; in general practice, mental health, and other disciplines where description is widely used, assuming the same word means the same thing is probably dangerous. However, in more formulaic data, such as pathology test results, claim reimbursement forms, and hospital discharges and referrals, the same code in two places can probably be more safely assumed to mean the same thing.

Accordingly, archetypes must at least provide a way of linking to external terminologies, even if they are not used as the primary definition of terms. The “binding” sections of the ontology section of an archetype provide this feature, enabling two kinds of correspondence to be stated:

- between a local `[atNNNN]` code and a (hopefully) equivalent code in an external terminology;
- between a local constraint code `[acNNNN]` and the result of a query to an external terminology;

Since bindings might be required to more than one terminology, the ADL syntax can be used to define each set of bindings on a per-terminology basis. Note that, as pointed out earlier, some or even most terms in an archetype might not be available in a particular terminology, so the size of a given binding might be much smaller than the set of primary local definitions in the archetype.

The overall result of the ADL approach is to enable archetypes to be self-defining, while including relationships to as many external terms as deemed reliable for computational use. This is most likely to be safest when interfacing with high-quality, targetted ontologies, such as for particular disease categories or treatment approaches.

7 The ADL Parsing Process

7.1 Overview

FIGURE 6 provides a graphical illustration of the ADL parsing process. ADL file is converted by the ADL parser into an ADL parse tree. This tree is an in-memory object structure representation of the semantics of the archetype, in a form convenient for further processing. The ADL tree is independent of reference models however, and needs to be processed further by an archetype builder, to create a valid archetype in object form for runtime use in a particular information system. As an input, the builder requires the specification of the reference information model - i.e. runtime access to the actual model of information whose instances will form the data of the final system. This might be via CASE tool files, XMI (XML model interchange files) or even validated programming language files.

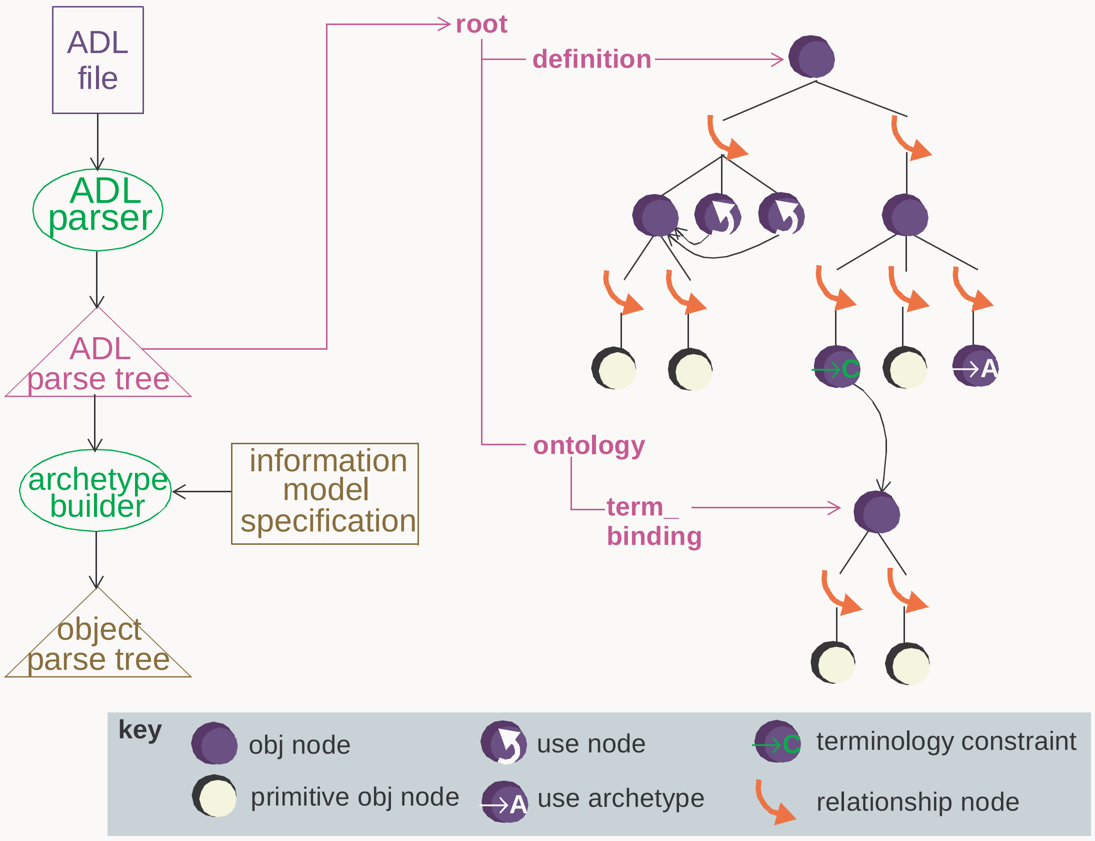


FIGURE 6 Parsed ADL Structure

The parse tree resulting from parsing an ADL file is shown on the right. It consists of alternate layers of object and relationship nodes, each containing the next level of nodes. At the leaves are leaf nodes - object nodes constraining primitive types such as `String`, `Integer` etc. There are also “use” nodes which represent internal references to other nodes, text constraint nodes which refer to a text constraint in the constraint binding part of the archetype, and archetype constraint nodes, which represent constraints on other archetypes allowed to appear at a given point. The full list of node types is as follows:

Object node: any interior node representing a constraint on instances of some type, e.g. `ENTRY`, `SECTION`;

Property node: a node representing any property (relationship or attribute) of an object type;

Leaf object node: an object node representing a constraint on a basic (built-in) object type;

Object reference node ("use" node): a node which refers to another object node already defined. The reference is made using a path;

Text constraint reference node: a node which refers to an object node defining a constraint on a plain text or coded term entity, which appears in the constraint binding section of the archetype; the reference is made using an `[acNNNN]` code;

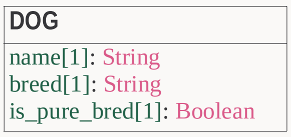
Archetype reference: a node whose statements define a constraint on other archetypes which are allowed to appear at that point in the archetype.

`To Be Continued:`

8 Predefined Type Libraries

8.1 Introduction

Standard ADL natively recognises only common primitive types, such as Integer, Real, String, Boolean, and data/time types. However, in each domain or subdomain in which a language like ADL might be used, there are certain commonly used types whose semantics are agreed by a large number of users. By *type* here, we mean a formal type of which data could be an instance, and about which a fragment of cADL could be written. Each such type has a UML model associated with it. For example, the type DOG might be described by the UML in FIGURE 7 below. Valid ADL statements based on this model are shown below; the first form using standard ADL, the second using a new syntax element (contrived for this example) representing DOG instances.



```
DOG[at0000] ∈ {
        name ∈ {"fido"}
        breed ∈ {"rottweiler"}              {$fido/rottweiler/pure$}
        is_pure_bred ∈ {True}
}
```

FIGURE 7 Example ADL Predefined Type

If within a domain or community of users, the above model of DOG is agreed, then similarly, ADL fragments consistent with this definition are also agreed in principle. In general, in a library of predefined types, there are type definitions, and ADL fragments. The latter usually follow the syntax of the main part of the language, but in some cases, may bring in small changes to the syntax, for efficiently representing constraint values in a way recognised by the community in question. The effect of providing special syntax is usually to remove one 'block' (enclosed by braces). An example of dedicated syntax is that for the class CODE_PHRASE in the following section; the use of the [terminology_id::code_phrase] syntax replaces the following lines of ADL:

```
CODE_PHRASE matches {
    terminology_id matches {"terminology_id"}
    code_string matches {"code_phrase"}
}
```

8.2 Implementing Type Libraries

To Be Continued: plug-in modules

8.3 Library Provenance

There are two ways to view standardised types:

- as an optional part of ADL, which can be used by anyone who finds them useful. In this case, changes to the definitions are done by the community maintaining of ADL itself;
- as an information model standardised by some body, in which case, the model is maintained by that body.

The following sections describe libraries of general-purpose predefined types in the first category.

9 Clinical ADL Predefined Type Library

9.1 Introduction

This section describes some predefined types and semantics common to science and clinical medicine. The type library is classified into the major categories terminological, quantitative, and date/time. For many of these types, there is the possibility of using a specific ADL syntax, in addition to the usual means of expression syntax provided by standard ADL.

9.2 Terminological Types

FIGURE 8 illustrates a set of text and coded text types typically used in health information systems. The types TEXT and CODED_TEXT represent respectively a text string in a given language, and a text string which has a corresponding code in some terminology or knowledge resource.

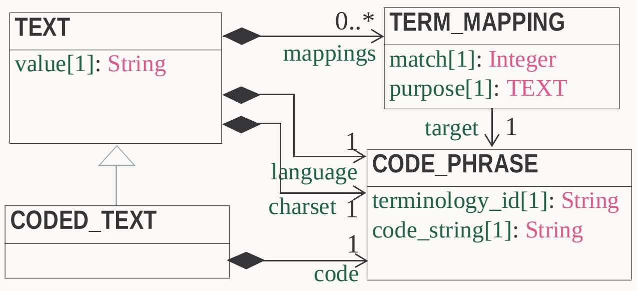


FIGURE 8 TEXT Types

The type CODE_PHRASE represents the actual code(s), which are assumed to have been generated by a terminology service of some kind. A code-phrase may be a single code, or a phrase expressing a coordination of codes which represents e.g. a word phrase in some language. The following subsections show how these types are constrained in ADL.

9.2.1 TEXT

Instances of the class TEXT can be constrained in standard cADL, using regular expressions. The main use of such expressions is to control the characters which can be used in various attributes such as person names; also to define patterns for things like patient identifiers, e.g.

```
pat_id matches {/[a-z]{1,3}-[0-9]{6,10}/}
```

9.2.2 CODED_TEXT

Items of CODED_TEXT can be constrained using standard cADL, as in the following pattern:

```
CODED_TEXT matches {
    code matches {} -- see below
    mappings matches {
        TERM_MAPPING matches {
            ...
        }
    }
}
```

9.2.3 CODE_PHRASE

9.2.3.1 Value Expressions

A CODE_PHRASE instance can be expressed in the form of an identifier of an external resource term code from an identified vocabulary. Terms are always enclosed in brackets ("[]"), and are either local to the archetype - i.e. of the form [local::atNNNN] or from some external vocabulary, in which case they take the form [TERMINOLOGY_ID::CODE], or [TERMINOLOGY_ID(version)::CODE]. Examples of terms are:

```
term = <[local::at0200]>
term = <[ICD10AM::F24]>
term = <[ICD10AM(2001)::F24]>
terms = <[ICD10AM::F24], [ICD10AM::F24]> -- List of ICD10AM codes
```

9.2.3.2 Constraint Expressions

In cADL, the *code* attribute from the CODED_TEXT example above is constrained using standard cADL as follows:

```
code matches {
    CODE_PHRASE matches {
        terminology_id matches {"xxxx"}
        code_string matches {"cccc"}
    }
}
```

or using two types of efficient, built-in literal expressions. Both avoid the need to state the class name. The first indicates that a CODE_PHRASE instance is constrained to a set which is the result of some interaction with a knowledge service; such expressions are stored in the ontology part of an archetype, and referenced with an [acNNNN] identifier, as follows:

```
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'")
```

The second kind of CODE_PHRASE constraint is one in which the terminology id, and actual code or set of codes forming the allowed set of values is given inline, as in the following examples:

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

code matches {[hl7_ClassCode::EVN, OBS]}

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

There is an important semantic difference between the two forms of constraint. The first approach says that the allowed set of values is known *outside* the archetype, in some other knowledge resource, while the second states that the allowed set is defined by the archetype itself - i.e. the archetype is the primary knowledge resource in this particular case.

9.2.4 Queries

The queries referred to above may be included in dADL data, e.g.:

```
    items("ac0001") = <query("terminology",
                         "terminology_id = ICD10 AND code matches 'J*'")>
```

This query is to an assumed "terminology" service in the environment (such as an implementation of the OMG HDTF Terminology Query Service, or the HL7 Clinical Terminology Service), and specifies that any term in ICD10 whose code matches the pattern 'J*' (any respiratory problem) should be returned.

All queries are assumed to return a `List<String>`; that is, it is assumed that the query interface will convert any return data, no matter how complex, into string form. The most typical example of this is when terminology "terms" are returned; they can be expressed as a string using the syntax described above, i.e. `"[TERMINOLOGY_ID::CODE]"`, e.g. `"[ICD10::J10]"`. Returned values can then be converted to a specific type, based on the type of the node containing the `"acNNNN"` reference.

A more generalised string format would be XML instance, and/or dADL (converted from XML instance or structured form).

To Be Determined: a standard way of expressing a code in a terminology has not yet been agreed by HL7, CEN. The syntax above is currently used in *open*EHR.

The only constraint on query syntax in dADL is that it follow the general form:

```
    query("service_name", "query text")
```

No assumption is made about the valid services or query syntaxes.

9.3 Quantitative Types

FIGURE 9 illustrates a partial model of a set of common quantitative types used in science and clinical medicine.

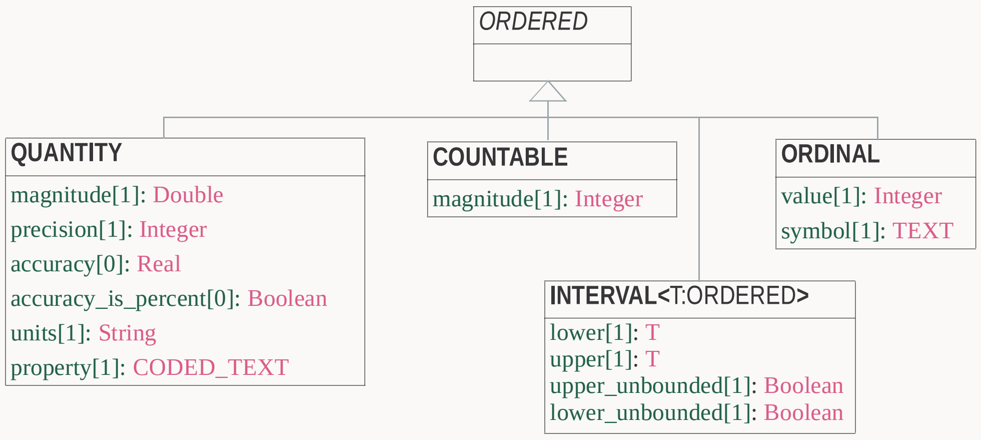


FIGURE 9 Quantitative Types

9.3.1 QUANTITY

To Be Determined: this section under construction

This type is used to represent measured continuous variables, and consists of a magnitude, units and property. Accuracy and precision can also be supplied if required. The following example shows a constraint corresponding to a blood pressure, expressed using any pressure unit.

```
    definition
        QUANTITY matches {
            magnitude matches {0.0..500.0}
            units matches {[ac0001]}
        }
    ontology
        ...
        items("ac0001") = <query("units", "unit matches 'FORCE/DISTANCE^2'")>
```

In the above, the expression "FORCE/DISTANCE^2" is an instance of a code phrase from a terminology called "units"; i.e. most likely a post-coordination from units term engine.

9.3.2 COUNTABLE

The COUNTABLE type is used to represent inherently integral things, such as “a number of steps”, “previous number of pregnancies” and so on. Countables have no units or property attributes, and are constrained using standard cADL, such as the following:

```
    previous_pregnancies matches {
        COUNTABLE matches {
            magnitude matches {0..20}
        }
    }
```

9.3.3 ORDINAL

An ordinal value is defined as one which is ordered without being quantified, and is represented by a symbol and an integer number. Ordinals can be constrained by standard cADL, although in a relatively lengthy way:

```
        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 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.

9.3.4 INTERVAL

Intervals of the ORDERED types are constrained using the standard ADL syntax element ".." between any two instances of a subtype of ORDERED.

To Be Continued:

9.4 Date/Time Types

FIGURE 10 illustrates a set of basic data/time types for use in clinical medicine. In addition to ADL's assumed primitive types of DATE, TIME, DATE_TIME and DURATION, three partial types are added, and a timezone attribute is added to the types TIME and DATE_TIME. All of these types are constrainable by the standard cADL date and time constraint statements (see Constraints on Dates, Times and Durations on page 40); whenever an "XX" or "??" is encountered in a constraint pattern, one of the partial types can be inferred.

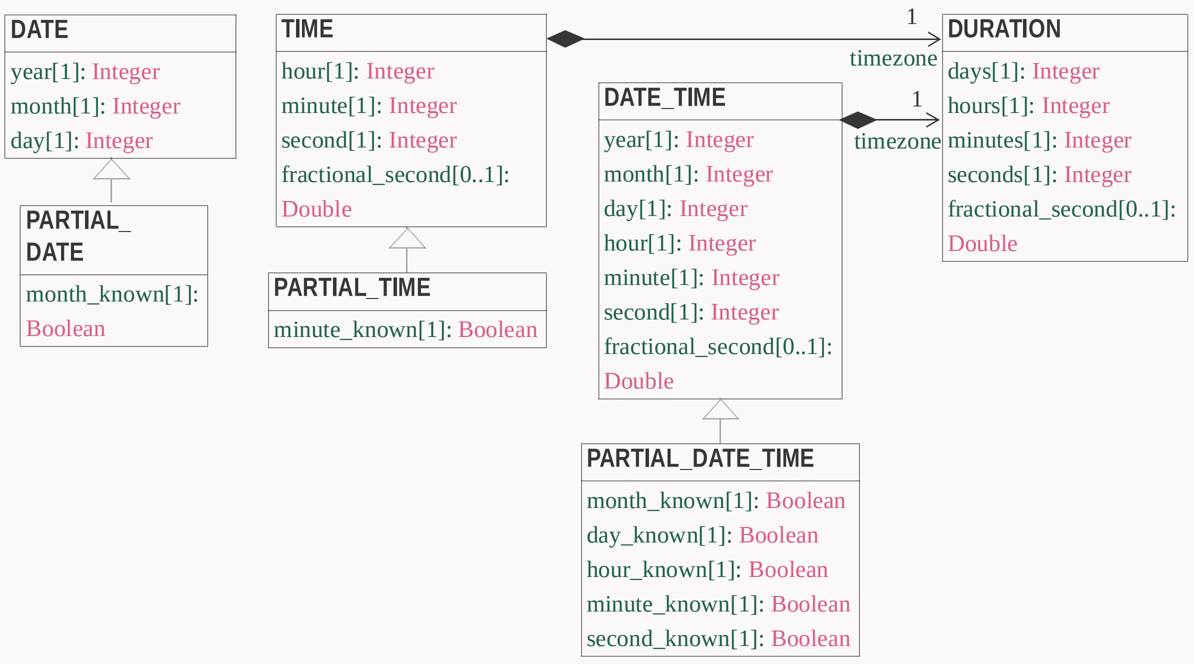


FIGURE 10 Date/Time Types

9.5 Additions to the dADL Model

The following figure shows changes to the dADL class model to accomodate the semantics described above. The only addition is that of the the DADL_OBJECT_TERM class.

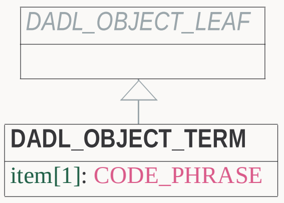


FIGURE 11 Additions to the dADL model

9.6 Additions to the cADL Model

The following figure shows changes to the cADL class model to accomodate the semantics described above. In fact, the only change is the addition of the `CADL_OBJECT_TERM` class.

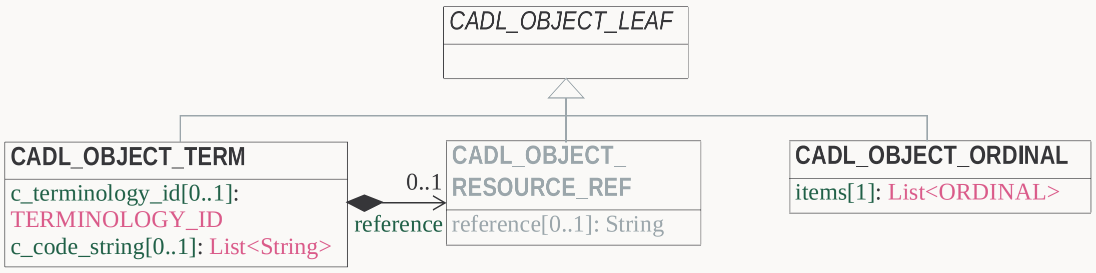


FIGURE 12 Additions to the cADL model

J 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. *A Shared Language for Archetypes and Templates - Part I*. 2003.
Available at http://www.deepthought.com.au/health/archetypes/archetype_language_2v0.6.2.doc.

4 Beale T, Heard S. *A Shared Language for Archetypes and Templates - Part II*. 2003.
Available at http://www.deepthought.com.au/health/archetypes/archetype_language.doc.

5 Beale T. *A Short Review of OCL*.
See http://www.deepthought.com.au/it/ocl_review.html.

6 Dolin R, Elkin P, Mead C *et al. HL7 Templates Proposal*. 2002.
Available at http://www.hl7.org.

7 Kilov H, Ross J. *Information Modelling: an Object-Oriented Approach*. Prentice Hall 1994.

8 Gruber T R. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. in Formal Ontology in Conceptual Analysis and Knowledge Representation. Eds Guarino N, Poli R. Kluwer Academic Publishers. 1993 (Aug revision).

9 Meyer B. *Eiffel the Language (2nd Ed)*. Prentice Hall, 1992.

10 Sowa J F. *Knowledge Representation: Logical, philosophical and Computational Foundations*. 2000, Brooks/Cole, California.

Resources

11 HL7 v3 RIM. See http://www.hl7.org.

12 HL7 Templates Proposal. See http://www.hl7.org.

13 Object Z. REF REQUIRED.

14 OWL - Web Ontology Language.
See http://www.w3.org/TR/2003/CR-owl-ref-20030818/.

15 *open*EHR. Knowledge-enabled EHR and related specifications.
See http://www.openEHR.org.

16 *open*EHR. EHR reference model.
See http://www.openEHR.org.

17 Perl Regular Expressions. REF REQUIRED.

18 Schematron. See http://www.ascc.net/xml/resource/schematron/schematron.html.

19 SynEx project, UCL. http://www.chime.ucl.ac.uk/HealthI/SynEx/.

END OF DOCUMENT