Archetype Definition Language (ADL)

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

Revision: 1.2draftC

Pages: 91

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

1. Ocean Informatics Australia

Copyright Notice

Amendment Record

Issue	Details	Who	Completed
1.2draftC	Added explanatory material; added domain type support; rewrite of most dADL sections. Rewrote OWL section based on input from: - Alan Rector, R Qamar (University of Manchester, UK), - I Román Martínez (University Seville, Spain). Various changes to assertions due to input from: - A Goodchild, - Z Z Tun (DSTC).	T Beale A Rector R Qamar I Román Martínez A Goodchild Z Z Tun	20 Sep 2004
1.2draftB	Detailed review by Eric Browne, Clinical Information Project, Australia. Remove UML models to "Archetype Object Model" document.	E Browne T Beale	20 July 2004
1.2draftA	CR-000103. Redevelop archetype UML model, add new keywords: allow_archetype, include, exclude. CR-000104. Fix ordering bug when use_node used. Required parser rules for identifiers to make class and attribute identifiers distinct. Added grammars for all parts of ADL, as well as new UML diagrams.	T Beale K Atalag	4 May 2004
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. Added cADL and dADL object models.	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

Trademarks

Table of Contents

1 Introduction

1.1 Purpose

This document describes the design basis and syntax of the Archetype Definition Language (ADL). It is intended for software developers, 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 by Beale [1], [2]. The *open*EHR Archetype Object Model [3] describes the model equivalent of the ADL syntax. The *open*EHR archetype framework is described in terms of Archetype Definitions and Principles [6] and an Archetype System [7].

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 structures and semantic constraints. Consequently, they bind terminology and ontology concepts to information model semantics, in order to make statements about what valid data 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 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.

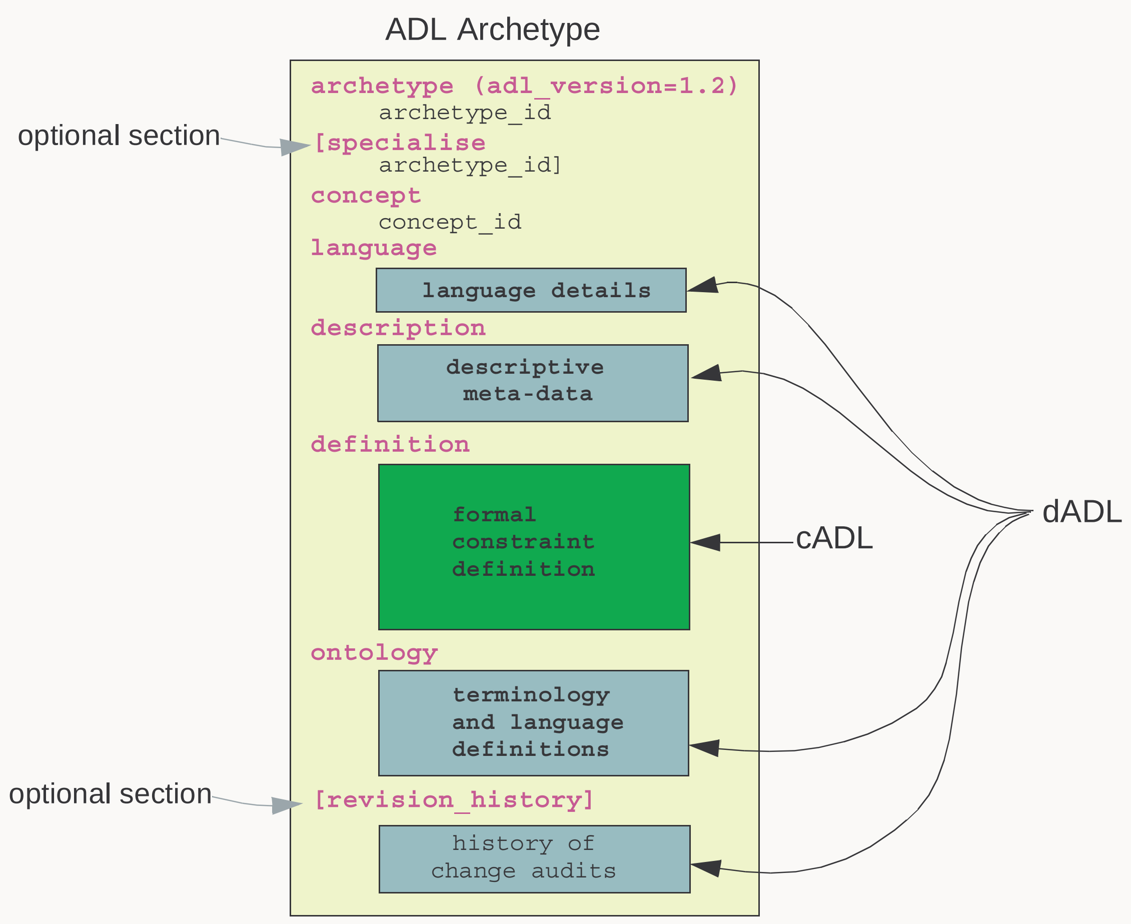


FIGURE 1 ADL Archetype Structure

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

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_version=1.2)
    adl-test-instrument.guitar.draft
concept
    [at0000] -- guitar
languages
    original_language = <"en">
    languages_available = <"en", ...>
definition
    INSTRUMENT[at0000] matches {
       size matches {|60..120|}                 -- size 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
            term_definitions = <
                ["en"] = <
                    items = <
                        ["at0000"] = <
                            text = <"guitar">;
                            description = <"stringed instrument">
                        >
                        ["at0001"] = <
                            text = <"neck">;
                            description = <"neck of guitar">
                        >
                        ["at0002"] = <
                            text = <"timber">;
                            description = <"straight, seasoned timber">
                        >
                        ["at0003"] = <
                            text = <"steel">;
                            description = <"stainless steel">
                        >
                        ["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 (see “The openEHR Archetype Object Model” document). 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.

Regardless of the possible object models, the ADL syntax remains constant as the primary abstract formalism for expressing archetypes, and as such, can be used directly for authoring and sharing archetypes. Other serialised forms are of course possible, including various flavours of XML.

1.3 Relationship to Other Semantic 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? Research to date has shown that in fact, no other formalism

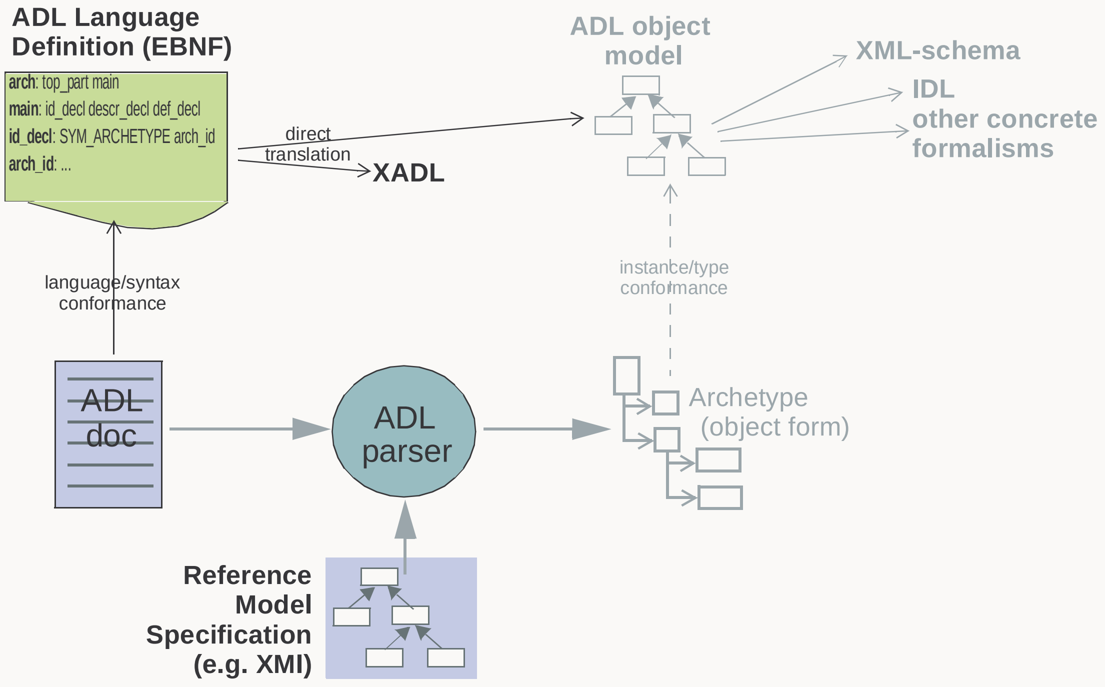

ADL Language Definition (EBNF)
arch: top_part main
main: id_decl descr_decl def_decl
id_decl: SYM_ARCHETYPE arch_id
arch_id: ...
direct translation
XADL
ADL object model
XML-schema
IDL
other concrete formalisms
language/syntax conformance
instance/type conformance
ADL doc
ADL parser
Archetype (object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

FIGURE 2 Relationship of ADL with Object Models

has been designed for the same use, and most do not easily express ADL semantics. When ADL was first being developed, it was felt that there was great value in analysing the problem space very carefully, and constructing an abstract syntax exactly matched to the solution, rather than attempting to use some other formalism - undoubtedly designed for a different purpose - to try and express the semantics of archetypes, or worse, to start with an XML-based exchange format, which often leads to the conflation of abstract and concrete representational semantics. Instead, the approach used has paid off, in that the resulting syntax is very simple and powerful, and in fact has allowed mappings to other formalisms to be more correctly defined and understood. The following sections compare ADL to other formalisms and show how it is different.

1.3.1 OWL (Web Ontology Language)

The Web Ontology Language (OWL) [20] is a W3C initiative for defining web-enabled ontologies which aim to allow the building of the "semantic web". OWL has an abstract syntax [12], developed at the University of Manchester, UK, and an exchange syntax, which is an extension of the XML-based syntax known as RDF (Resource Description Framework). We discuss OWL only in terms of its abstract syntax, since this is a semantic representation of the language unencumbered by XML or RDF details (there are tools which convert between abstract OWL and various exchange syntaxes).

OWL is a general purpose ontology language, and is primarily used to describe "classes" of things in such a way as to support *subsumptive* inferencing within the ontology, and by extension, on data which are instances of ontology classes. There is no general 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 are therefore usually constraint statements on an *implied* model on which data *appears* to be based. However, the semantics of an information model can themselves be represented in OWL. Restrictions are the primary way of defining subclasses.

In intention, OWL is aimed at representing some 'reality' and then making inferences about it; for example in a medical ontology, it can infer that a particular patient is at risk of ischemic heart disease due to smoking and high cholesterol, if the knowledge that ‘ischemic heart disease has-risk-factor smoking’ and ‘ischemic heart disease has-risk-factor high cholesterol’ are in the ontology, along with a representation of the patient details themselves. OWL’s inferencing works by subsumption, which is to say, asserting either that an ‘individual’ (OWL’s equivalent of an object-oriented instance or a type) conforms to a ‘class’, or that a particular ‘class’ ‘is-a’ (subtype of another) ‘class’; this approach can also be understood category-based reasoning or set-containment.

ADL can also be thought of as being aimed at describing a reality, and allowing inferences to be made. However, the ‘reality’ it describes are in terms of constraints on information structures (based on an underlying reference information model), and the inferencing is between data and the constraints. Some of the differences between ADL and OWL include the following.

- ADL syntax is predicated on the existence of existing object-oriented reference models, expressed in UML or some similar formalism, and the constraints in an ADL archetype are in relation to types and attributes from such a model. In contrast, OWL is far more general, and requires the explicit expression of a reference model in OWL, before archetype-like constraints could be expressed.
- Because information structures are in general hierarchical compositions of nodes and elements, and may be quite deep, ADL enables constraints to be expressed in a structural, nested way, mimicking the tree-like nature of the data it constrains. OWL does not provide a native way to do this, and although it is possible to express approximately the same constraints in OWL, it is fairly inconvenient, and would probably only be made easy by machine conversion from a visual format more or less like ADL.
- As a natural consequence of dealing with heavily nested structures in a natural way, ADL also provides a path syntax, based on Xpath [21], enabling any node in an archetype to be referenced by a path or path pattern. OWL does not provide an inbuilt path mechanism.
- ADL also natively takes care of disengaging natural language and terminology issues from constraint statements by having a separate ontology per archetype, which contains ‘bindings’ and language-specific translations. OWL has no inbuilt syntax for this, requiring such semantics to be represented from first principles.
- Lastly, OWL (as of mid 2004) is still under development, and has only a very limited set of primitive constraint types (it is not even possible to state a constraint on an Integer attribute of the form ‘any value between 80 and 110’), although this is being addressed; by constrast, ADL provides a rich set of constraints on primitive types, including dates and times.

Research to date shows that the semantics of an archetype might be representable inside OWL. To do so would require the following steps:

- express the relevant reference models in OWL (this has been shown to be possible);
- express the relevant terminologies in OWL (research on this is ongoing);
- be able to represent concepts (i.e. constraints) independently of natural language (status unknown);
- convert the cADL part of an archetype to OWL; assuming the problem of primitive type constraints is solved, research to date shows that this should in principle be possible.

To *use* the archetype on data, the data themselves would have to be converted to OWL, i.e. be expressed as ‘individuals’. In conclusion, we can say that mathematical equivalence between OWL and ADL is probably provable. However, it is clear that OWL is far from a convenient formalism to express archetypes, or to use them for modelling or reasoning against data. The ADL approach makes use of existing UML semantics and existing terminologies, and adds a convenient syntax for expressing the required constraints. It also appears fairly clear that even if all of the above conversions were achieved, using OWL-expressed archetypes to validate data (which would require massive amounts of data to be converted to OWL statements) is unlikely to be anywhere near as efficient as doing it with archetypes expressed in ADL or one of its concrete expressions.

Nevertheless, OWL provides a very powerful generic reasoning framework, and offers a great deal of inferencing power of far wider scope than the specific kind of 'reasoning' provided by archetypes. It appears that it could be useful for the following archetype-related purposes:

- providing access to ontological resources while authoring archetypes, including terminologies, pure domain-specific ontologies, etc;
- providing a semantic 'indexing' mechanism allowing archetype authors to find archetypes relating to specific subjects (which might not be mentioned literally within the archetypes);
- providing inferencing on archetypes in order to determine if a given archetype is subsumed within another archetype which it does not specialise (in the ADL sense);
- providing access to archetypes from within a semantic web environment, such as an ebXML server or similar.

Research on these areas is active in the US, UK, Australia and Spain.

1.3.2 OCL (Object Constraint Language)

The OMG's Object Constraint Language (OCL) appears to be an obvious contender for writing archetypes. However, its designed use is to write constraints on *object models*, rather than on *data*, which is what archetypes are about. As a concrete example, OCL can be used to make statements about the *actors* attribute of a class `Company` - e.g. that *actors* must exist and contain the `Actor` who is the *lead* of `Company`. However, it cannot describe the notion that for a particular kind of (acting) company, such as 'itinerant jugglers', there must be at least four actors, each of whom have among their *capabilities* 'advanced juggling', plus an Actor who has *skill* 'musician'. ADL provides the ability to create numerous archetypes, each describing in detail a concrete type of `Company`.

OCL's constraint types include function pre- and post-conditions, and class invariants. There is no structural character to the syntax - all statements are essentially first-order predicate logic statements about elements in models expressed in UML, and are related to parts of a model by 'context' statements. This makes it impossible to use OCL to express an archetype in a structural way which is natural to domain experts. OCL also has some flaws, described by Beale [4].

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 minor 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.

1.3.3 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 [9]) 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. Thus, to use KIF for expressing archetypes, a similar process as for OWL would have to be used, i.e. the existing reference model would have to be converted to KIF statements, along with terminologies, before archetypes themselves could be expressed.

It should also be realised that KIF is intended as a knowledge exchange format, rather than a knowledge representation format, which is to say that it can (in theory) represent the semantics of any other knowledge representation language, such as OWL. This distinction today seems fine, since web-enabled languages like OWL probably don't need an exchange format other than their XML equivalents to be shared. The relationship and relative strengths and deficiencies is explored by e.g. Martin [10].

1.4 XML Exchange Formats

Various concrete XML exchange formats can be defined based on XML, as described below.

1.4.1 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.

1.4.2 RDF

Raw RDF can be generated from abstract OWL, using various open source tools currently available.

1.5 Changes From Previous Versions

For existing users of ADL or archetype development tools, the following provides a guide to the changes in the syntax.

1.5.1 Version 1.2 from Version 1.1

Adl Version

The ADL version is now optionally (for the moment) included in the first line of the archetype, as follows.

```
    archetype (adl_version=1.2)
```

It is strongly recommended that all tool implementors include this information when archetypes are saved, enabling archetypes to gradually become imprinted with their correct version, for more reliable later processing. The `adl_version` indicator is likely to become mandatory in future versions of ADL.

dADL Syntax Changes

The dADL syntax for container attributes has been altered to allow paths and typing to be expressed more clearly, as part of enabling the use of Xpath-style paths. ADL 1.1 dADL had the following appearance:

```
    school_schedule = <
        locations(1) = <...>
        locations(2) = <...>
        locations(3) = <...>
        subjects("philosophy:plato") = <...>
        subjects("philosophy:kant") = <...>
        subjects("art") = <...>
    >
```

This has been changed to look like the following:

```
    school_schedule = <
        locations = <
            [1] = <...>
            [2] = <...>
            [3] = <...>
        >
        subjects = <
            ["philosophy:plato"] = <...>
            ["philosophy:kant"] = <...>
            ["art"] = <...>
        >
```

The new appearance both corresponds more directly to the actual object structure of container types, and has the property that paths can be constructed by directly reading identifiers down the backbone of any subtree in the structure. It also allows the addition of typing information anywhere in the structure, as shown in the following example:

```
    school_schedule = SCHEDULE <
        locations = LOCATION <
            [1] = <...>
            [2] = <...>
            [3] = ARTS_PAVILLION <...>
        >
        subjects = <
            ["philosophy:plato"] = ELECTIVE_SUBJECT <...>
            ["philosophy:kant"] = ELECTIVE_SUBJECT <...>
            ["art"] = MANDATORY_SUBJECT <...>
        >
```

These changes will affect the parsing of container structures and keys in the description and ontology parts of the archetype.

Revision History

Revision history is now recorded in a separate section of the archetype, both to logically separate it from the archetype descriptive details, and to facilitate automatic processing by version control systems in which archtypes may be stored. This section is included at the end of the archetype because it is in general a monotonically growing section.

Primary_language and Languages_available

An attribute previously called 'primary_language' was required in the ontology section of an ADL 1.1 archetype. This is renamed to 'original_language' and is now moved to a new top level section in the archetype called 'language'. Its value is still expressed as a dADL String attribute. The 'languages_available' attribute previously required in the ontology section of the archetype is also moved to this new top level section.

1.6 Tools

A validating ADL parser is freely available from http://www.openEHR.org. It has been wrapped for use in Java environments, in dotNet, and standard C/C++ environments. See the website for the latest status.

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 = 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">
        >
    >
    [01235] = <
        -- etc
    >
>
```

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. The UML semantics of composition/aggregation and association are expressible, as are shared objects. 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 81.

A common question about dADL is why it is needed, when there is already XML? This question highlights the widespread misconception about XML, namely that because it is can be read by a text editor, it is intended for humans. In fact, XML is designed for machine processing, and is textual to guarantee its interoperability. Realistic examples of XML (e.g. XML-schema instance, OWL-RDF ontologies) are generally unreadable for humans. dADL is on the other hand designed as a human-writable and readable formalism, which is also machine processable; it may be thought of as an *abstract syntax for XML instance* which conforms to one of the XML schema languages. dADL also differs from XML by:

- providing a more comprehensive set of leaf data types, including intervals of numerics and date/time types, and lists of all primitive types;
- adhering to object-oriented semantics, which XML schema languages generally do not;

- not using the confusing XML notion of ‘attributes’ and ‘elements’ to represent what are essentially object properties.

Of course, this does not prevent XML exchange syntaxes being used for dADL, and indeed the conversion to XML instance is rather straighforward. Details on the XML expression of dADL and use of Xpath expressions is descibed in section 2.6 on page 29 and section 2.7 on page 30.

2.2 Basics

2.2.1 Scope of a dADL Document

To Be Continued: 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.

2.2.2 Keywords

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

2.2.3 Comments

In a dADL text, comments satisfy the following rule:

> **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.

2.2.4 Quoting

The dADL quoting rule is as follows:

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

2.2.5 Information Model Identifiers

Two types of identifiers from information models are used in dADL: type names and attribute names.

> A **type name** is any identifier with an initial upper case letter, followed by any combination of letters, digits, and underscores. An **attribute name** is any identifier with an initial lower case letter, followed by any combination of letters, digits and underscores.

Type names in this document are 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`, as long as they obey the rule above. 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.

2.2.6 Instance Node Identifiers

Any node in a dADL text which is introduced by a type name can also be identified at the instance level, according to the following rule:

> **Instance nodes can be identified** in dADL using an identifier delimited by brackets, e.g. `[some_id]`. Any dADL identifier 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 this document in magenta.

2.2.7 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:

```
term = <text = <"plan">; description = <"The clinician's advice">>

term = <text = <"plan"> description = <"The clinician's advice">>

term = <
    text = <"plan">
    description = <"The clinician's advice">
>
```

> **Semi-colons** are completely optional.

2.3 Paths

The notion of paths is integral to using dADL data. Because dADL data is hierarchical, and all nodes are uniquely identified, reliable paths can be determined to every node in an dADL text. 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, indicated by brackets (‘[]’). A path either finishes in a slash, and identifies an object node, or finishes in a relationship name, and identifies a relationship node. A typical path in dADL is as follows:

```
/term_definitions[en]/items[at0001]/text/
```

In the following sections, paths are explained for all the dADL data examples.

2.4 Structure

2.4.1 General Form

A dADL document or text is always a purely hierarchical expression of an instance of an object or entity. In its simplest form, a dADL text consists of repetitions of the following pattern:

```
attribute_id = <value>
```

In the most basic form of dADL, each attribute id is the name of an attribute in an implied or real object or relational model of data. Each “value” is either a literal value of a primitive type (see Primitive Types on page 27) or a further nesting of attribute names and values, terminating in leaf nodes of primitive type values. Where sibling attribute nodes occur, the attribute identifiers must be unique, just as in a standard object or relational model.

Sibling attribute identifiers must be unique.

The following shows a typical structure.

```
aaa = <
    bbbbb = <
       xx = <leaf_value>
       yy = <leaf_value>
    >
    ccc = <
       pppp = <
          xx = <leaf_value>
       >
       qq = <leaf_value>
    >
>
ddd = <>
```

In the above structure, each "<>" encloses an instance of some type. The hierarchical structure corresponds to the part-of relationship between objects, otherwise known as *composition* and *aggregation* relationships in object-oriented formalisms such as UML (the difference between the two is usually described as being "sub-objects related by aggregation can exist on their own, whereas sub-objects related by composition are always destroyed with the parent"; dADL does not differentiate between the two, since it is the business of a model, not the data, to express such semantics).

2.4.1.1 Outer Delimiters

To be completely regular, an outer level of delimiters should be used, because the totality of a dADL text is an object, not a collection of disembodied attribute/object pairs. However, the outermost delimiters can be left out in order to improve readability, and without complicating the parsing process. The completely regular form would appear as follows:

```
<
    aaa = <
    >
    ddd = <>
>
```

Outer '<>' delimiters in a dADL text are optional.

2.4.1.2 Paths

The complete set of paths for the above example is as follows.

```
aaa
aaa/bbbbb
aaa/bbbbb/xx/       -- path to a leaf value
aaa/ccc
aaa/ccc/pppp/
aaa/ccc/pppp/xx     -- path to a leaf value
aaa/ccc/qq/         -- path to a leaf value
ddd
```

2.4.2 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.

Note: within this document, empty sections are shown in many places to represent fully populated data, which would of course require much more space.

Empty sections can appear anywhere.

2.4.3 Container Objects

The syntax described so far allows an instance of an arbitrarily large object to be expressed, but does not yet allow for attributes of container types such as lists, sets and hash tables, i.e. items whose type in an underlying reference model is something like `attr:List<Type>`, `attr:Set<Type>` or `attr: Hash<ValueType, KeyType>`. There are two ways instance data of such container objects can be expressed. The first is to use a list style literal value, where the “list nature” of the data is expressed within the manifest value itself, as in the following examples.

```
fruits = <“pear”, “cumquat”, “peach”>
some_primes = <1, 2, 3, 5>
```

See Lists of Primitive Types on page 28 for the complete description of list leaf types. This approach is fine for leaf data. However for containers holding non-primitive values, including more container objects, the addition of *qualified explicit container member attributes* is made. We can imagine that an instance of a container could be expressed as follows.

```
-- WARNING: THIS IS NOT VALID dADL
people = <
    <name = <> date_of_birth = <> sex = <> interests = <>>
    <name = <> date_of_birth = <> sex = <> interests = <>>
    -- etc
>
```

Here, “anonymous” blocks of data are repeated within the outer block. However, this makes the data hard to read, and does not make it easy to understand how paths would be constructed through the data. Consequently, dADL takes a different approach, exemplified by the following.

```
people = <
    [“akmal:1975-04-22”] = <name = <> birth_date = <> interests = <>>
    [“akmal:1962-02-11”] = <name = <> birth_date = <> interests = <>>
    [“gianni:1978-11-30”] = <name = <> birth_date = <> interests = <>>
>
```

For each contained object, we use the name of the container attribute “people” with a qualifier, or *key*, to ensure uniqueness; in the case above, strings are used. Qualifiers can be of any basic comparable type, and follow exactly the same syntax described below for data of primitive types. Usually, qualifier values are constructed from one or more of the values of the contained items, in the same way as relational database keys are constructed from sufficient field values to guaranteed uniqueness. However, they need not be - they may be independent of the contained data, as in the case of hash tables, where the keys are part of the hash table structure, or equally, they may simply be integer index values, as in the ‘locations’ attribute in the ‘school_schedule’ structure shown below.

Container structures can of course appear anywhere in an overall instance structure, allowing complex data such as the following to be expressed in a readable way.

```
school_schedule = <
    lesson_times = <08:30:00, 09:30:00, 10:30:00, ...>

    locations = <
       [1] = <"under the big plane tree">
       [2] = <"under the north arch">
       [3] = <"in a garden">
    >

    subjects = <
       ["philosophy:plato"] = < -- note construction of qualifier
          name = <"philosophy">
          teacher = <"plato">
          topics = <"meta-physics", "natural science">
          weighting = <76%>
       >
       ["philosophy:kant"] = <
          name = <"philosophy">
          teacher = <"kant">
          topics = <"meaning and reason", "meta-physics", "ethics">
          weighting = <80%>
       >
       ["art"] = <
          name = <"art">
          teacher = <"goya">
          topics = <"technique", "portraiture", "self-mockery">
          weighting = <78%>
       >
    >
```

Container instances are expressed using repetitions of a block introduced by an arbitrary container attribute name, qualified in each case by a unique manifest value.

2.4.3.1 Paths

Paths through container objects are formed in the same way as paths in other structured data, with the addition of the key, to ensure uniqueness. The key is included syntactically enclosed in brackets, in a similar way to how keys are included in Xpath expressions. Paths through containers in the above example include the following:

```
school_schedule/locations[1]/              -- path to "under the big..."
school_schedule/subjects["philosophy:kant"]/-- path to "kant"
```

2.4.4 Nested Container Objects

To Be Continued:

2.4.5 Adding Type Information

In many cases, dADL data is of a simple structure, very regular, and highly repetitive, such as the expression of simple demographic data. In such cases, it is preferable to express as little as possible about the implied reference model of the data (i.e. an object or relational model from to which it conforms), since various software components want to use the data, and use it in different ways. However, there are also cases where the data is highly complex, and more model information is needed to help software parse it. Examples include large design databases, such as for aircraft, and health records. Typing information is added to instance data simply by including the type name after the '=' sign, as in the following example.

```
destinations = <
    ["seville"] = TOURIST_DESTINATION <
        profile = DESTINATION_PROFILE <>
        hotels = <
            ["gran sevilla"] = HISTORIC_HOTEL <>
            ["sofitel"] = LUXURY_HOTEL <>
            ["hotel real"] = PENSION <>
        >
        attractions = <
            ["la corrida"] = ATTRACTION <>
            ["Alcázar"] = HISTORIC_SITE <>
        >
    >
>
```

Note that in the above, no type identifiers are included after the "hotels" and "attractions" attributes, and it is up to the processing software to infer the correct types (usually easy - it will be pre-determined by an information model). However, the complete typing information can be included, as follows.

```
        hotels = List<HOTEL> <
            ["gran sevilla"] = HISTORIC_HOTEL <>
        >
```

This illustrates the use of generic, or "template" type identifiers, expressed in the standard UML syntax, using angle brackets. Any number of template arguments and any level of nesting is allowed, as in the UML. There is a small risk of visual confusion between the template type delimiters and the standard dADL block delimiters, but technically there can never be any confusion, because only type names (first letter capitalised) may appear inside template delimiters, while only attribute names (first letter lower case) can appear after a dADL block delimiter.

To Be Determined: is it better to allow just "HOTEL" rather than "List<HOTEL>"?

Type identifiers can also include namespace information, which is necessary when same-named types appear in different packages of a model. Namespaces are included by prepending package names, separated by the '.' character, in the same way as in most programming languages, as in the qualified type names **`RM.EHR.CONTENT.ENTRY`** and **`Core.Abstractions.Relationships.Relationship`**.

> **Type Information** can be included optionally on any node immediately before the opening '<' of any block, in the form of a UML-style type identifier which optionally includes dot-separated namespace identifiers and template parameters.

2.4.6 Associations and Shared Objects

All of the facilities described so far allow any object-oriented data to be faithfully expressed in a formal, systematic way which is both machine- and human-readable, and allow any node in the data to be addressed using an Xpath-style path. The availability of reliable paths allows not only the representation of single 'business objects', which are the equivalent of UML *aggregation* (and *composition*) hierarchies, but the also representation of *associations* between objects, and by extension, shared objects.

Consider that in the example above, 'hotel' objects may be shared objects, referred to by assocation. This can be expressed as follows.

```
    destinations = <
        ["seville"] = <
            hotels = <
                ["gran sevilla"] = </hotels["gran sevilla"]>
                ["sofitel"] = </hotels["sofitel"]>
                ["hotel real"] = </hotels["hotel real"]>
            >
        >
    >

    bookings = <
        ["seville:0134"] = <
            customer_id = <"0134">
            period = <...>
            hotel = </hotels["sofitel"]>
        >
    >

    hotels = <
            ["gran sevilla"] = HISTORIC_HOTEL <>
        ["sofitel"] = LUXURY_HOTEL <>
        ["hotel real"] = PENSION <>
    >
```

Associations are expressed via the use of fully qualified paths as the data for a attribute. In this example, there are references from a list of destinations, and from a booking list, to the same hotel object. If type information is included, it should go in the declarations of the relevant objects; type declarations can also be used before path references, which might be useful if the association type is an ancestor type of the type of the actual object being referred to.

2.4.6.1 Paths

The path set from the above example is as follows:

```
    /destinations["seville"]/hotels["gran sevilla"]/
    /destinations["seville"]/hotels["sofitel"]/
    /destinations["seville"]/hotels["hotel real"]/

    /bookings["seville:0134"]/customer_id/
    /bookings["seville:0134"]/period/
    /bookings["seville:0134"]/hotel/

    /hotels["sofitel"]/
    /hotels["hotel real"]/
    /hotels["gran sevilla"]/
```

2.5 Leaf Data

All dADL data eventually devolve to instances of the primitive types `String`, `Integer`, `Real`, `Double`, `String`, `Character`, various date/time types, lists or intervals 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>
```

2.5.1 Primitive Types

2.5.1.1 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
```

All characters are case-sensitive, i.e. ‘a’ is distinct from ‘A’.

2.5.1.2 String Data

All strings are enclosed in double quotes, 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
```

All strings are case-sensitive, i.e. ‘word’ is distinct from ‘Word’.

2.5.1.3 Integer Data

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

```
25
300000
29e6
```

Commas or periods for breaking long numbers are not allowed, since they confuse the use of commas used to denote list items (see section 2.5.3 below).

2.5.1.4 Real Data

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

```
25.0
3.1415926
6.023e23
```

Commas or periods for breaking long numbers are not allowed. Only periods may be used to separate the decimal part of a number; unfortunately, the European use of the comma for this purpose conflicts with the use of the comma to separete list items (see section 2.5.3 below).

2.5.1.5 Boolean Data

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

```
True
False
```

2.5.1.6 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
```

2.5.2 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 $\pm$ 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
```

2.5.3 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], ...
```

White space may be freely used or avoided in lists, i.e. the following two lists are identical:

```
1,1,2,3
1, 1, 2,3
```

2.6 Data Navigation

Characteristics of the Syntax

The dADL syntax as described above has a number of useful characteristics which enable the extensive use of paths to navigate it, and express references. These include:

- each <> block corresponds to an object (i.e. an instance of some type in an information model);
- the name before an '=' is always an attribute name or else a container element key, which attaches to the attribute of the enclosing block;
- paths can be formed by simply running down a tree branch and concatenating attribute name, container keys (where they are encountered) and '/' characters;
- every node is reachable by a path;
- shared objects can be referred to by path references.

Comparison with Xpath

The dADL path syntax is a subset of the Xpath query language, with a few shortcuts added to reduce the verbosity of the most common cases. 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 dADL, 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:

```
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'
```

The Xpath equivalents are:

```
items[@id=1]      -- the member of 'items' with key '1'
items[@id="foo"]-- the member of 'items' with key 'foo'
items[@id=at0001]-- the member of 'items' with key 'at0001'
```

To Be Continued:

2.7 Expression of dADL in XML

The dADL syntax maps quite easily to XML instance. It is important to realise that people using XML often develop different mappings for object-oriented data, due to the fact that XML does not have systematic object-oriented semantics. This is particularly the case where containers such as lists and sets such as 'employees: List<Person>' are mapped to XML; many implementors have to invent additional tags such as 'employee' to make the mapping appear visually correct. The particular mapping chosen here is designed to be a faithful reflection of the semantics of the object-oriented data, and does not take into account visual aesthetics of the XML (since it is a machine syntax not intended for direct reading by humans). The result is that Xpath expressions are the same for dADL and XML, and also correspond to what one would expect based on an underlying object model.

The main elements of the mapping are as follows.

- single attribute nodes map to tagged nodes of the same name
- container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute 'id' set to the dADL key. For example, the dADL:

```
subjects = <
   ["philosophy:plato"] = <
      name = <"philosophy">
   >
   ["philosophy:kant"] = <
      name = <"philosophy">
   >
>
```

  maps to the XML:

```
<subjects id="philosophy:plato">
      <name>
         philosophy
      </name>
</subjects>
<subjects id="philosophy:kant">
      <name>
         philosophy
      </name>
</subjects>
```

  This guarantees that the path `subjects[@id="philosophy:plato"]/name` navigates to the same element in both dADL and the XML.

- type names map to XML 'type' attributes e.g. the dADL

```
destinations = <
   ["seville"] = TOURIST_DESTINATION <
      profile = DESTINATION_PROFILE <>
      hotels = <
         ["gran sevilla"] = HISTORIC_HOTEL <>
      >
   >
>
```

  maps to:

```
<destinations id="seville" type=TOURIST_DESTINATION>
      <profile type=DESTINATION_PROFILE>
         ...
      </profile>
      <hotels id="gran sevilla" type=HISTORIC_HOTEL>
```

```
                    ...
                </hotels>
            >
        >
```

2.8 dADL Syntax

2.8.1 Grammar

This section shows the dADL grammar, as implemented and tested in the *open*EHR implementation project.

```
    --
    -- dADL grammar @changeset 1.24, openEHR implem-dev BK repository
    --

    input: attr_vals -- anonymous object
        | identified_object
        | error
        ;

    identified_object: identified_object_head SYM_EQ SYM_START_DBLOCK attr_vals
                       SYM_END_DBLOCK
        ;

    identified_object_head: V_IDENTIFIER V_LOCAL_TERM_CODE_REF
        ;

    attr_vals: attr_val
        | attr_vals attr_val
        | attr_vals ';' attr_val
        ;

    attr_val: attr_id SYM_EQ attr_val_body
        ;

    attr_id: V_IDENTIFIER
        | V_IDENTIFIER '(' simple_value ')'
        | V_IDENTIFIER '(' error
        ;

    attr_val_body: object_empty
        | object_complex_head object_complex_body
        | object_basic
        ;

    object_complex_head: SYM_START_DBLOCK
        ;

    object_complex_body: attr_vals SYM_END_DBLOCK
        ;

    object_basic: SYM_START_DBLOCK basic_object_val SYM_END_DBLOCK
        ;

    object_empty: SYM_START_DBLOCK SYM_END_DBLOCK
        ;
```

```
basic_object_val: simple_value
     | simple_list_value
     | simple_interval_value
     | term_code
     | term_code_list_value
     | query
     ;

query: SYM_QUERY_FUNC '(' V_STRING ',' V_STRING ')'
     ;

simple_value: string_value
     | integer_value
     | real_value
     | boolean_value
     | character_value
     | date_value
     | time_value
     | date_time_value
     | duration_value
     ;

simple_list_value: string_list_value
     | integer_list_value
     | real_list_value
     | boolean_list_value
     | character_list_value
     | date_list_value
     | time_list_value
     | date_time_list_value
     | duration_list_value
     ;

simple_interval_value: integer_interval_value
     | real_interval_value
     | date_interval_value
     | time_interval_value
     | date_time_interval_value
     | duration_interval_value
     ;

---------------------- BASIC DATA VALUES -----------------------

string_value: V_STRING
     ;

string_list_value: V_STRING ',' V_STRING
     | string_list_value ',' V_STRING
     | V_STRING ',' SYM_LIST_CONTINUE
     ;

integer_value: V_INTEGER
     | '+' V_INTEGER
     | '-' V_INTEGER
     ;
```

```
integer_list_value: integer_value ',' integer_value
     | integer_list_value ',' integer_value
     | integer_value ',' SYM_LIST_CONTINUE
     ;

integer_interval_value: SYM_INTERVAL_DELIM integer_value SYM_ELLIPSIS
          integer_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_LT integer_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_LE integer_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_GT integer_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_GE integer_value SYM_INTERVAL_DELIM
     ;

real_value: V_REAL
     | '+' V_REAL
     | '-' V_REAL
     ;

real_list_value: real_value ',' real_value
     | real_list_value ',' real_value
     | real_value ',' SYM_LIST_CONTINUE
     ;

real_interval_value: SYM_INTERVAL_DELIM real_value SYM_ELLIPSIS real_value
                       SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_LT real_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_LE real_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_GT real_value SYM_INTERVAL_DELIM
     | SYM_INTERVAL_DELIM SYM_GE real_value SYM_INTERVAL_DELIM
     ;

boolean_value: SYM_TRUE
     | SYM_FALSE
     ;

boolean_list_value: boolean_value ',' boolean_value
     | boolean_list_value ',' boolean_value
     | boolean_value ',' SYM_LIST_CONTINUE
     ;

character_value: V_CHARACTER
     ;

character_list_value: character_value ',' character_value
     | character_list_value ',' character_value
     | character_value ',' SYM_LIST_CONTINUE
     ;

date_value: V_INTEGER '-' V_INTEGER '-' V_INTEGER
     ; -- in ISO8601 form yyyy-MM-dd


date_list_value: date_value ',' date_value
     | date_list_value ',' date_value
     | date_value ',' SYM_LIST_CONTINUE
     ;
```

```
date_interval_value: SYM_INTERVAL_DELIM date_value SYM_ELLIPSIS date_value
                     SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LT date_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LE date_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GT date_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GE date_value SYM_INTERVAL_DELIM
    ;

time_value: time_point
    | time_point time_zone
    ;

time_point: V_INTEGER ':' V_INTEGER ':' V_INTEGER
    | V_INTEGER ':' V_INTEGER ':' V_REAL
    ;

time_zone: 'Z'
    | '+' V_INTEGER
    ;

time_list_value: time_value ',' time_value
    | time_list_value ',' time_value
    | time_value ',' SYM_LIST_CONTINUE
    ;

time_interval_value: SYM_INTERVAL_DELIM time_value SYM_ELLIPSIS time_value
                      SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LT time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LE time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GT time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GE time_value SYM_INTERVAL_DELIM
    ;

date_time_value: date_value time_value
    ;

date_time_list_value: date_time_value ',' date_time_value
    | date_time_list_value ',' date_time_value
    | date_time_value ',' SYM_LIST_CONTINUE
    ;

date_time_interval_value: SYM_INTERVAL_DELIM date_time_value SYM_ELLIPSIS
                          date_time_value  SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LT date_time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_LE date_time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GT date_time_value SYM_INTERVAL_DELIM
    | SYM_INTERVAL_DELIM SYM_GE date_time_value SYM_INTERVAL_DELIM
    ;

duration_value: duration_magnitude
    | '-' duration_magnitude
    ;

duration_magnitude: V_ISO8601_DURATION
    ;

duration_list_value: duration_value ',' duration_value
```

```
        | duration_list_value ’,’ duration_value
        | duration_value ’,’ SYM_LIST_CONTINUE
        ;

    duration_interval_value: SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS
                             duration_value SYM_INTERVAL_DELIM
        | SYM_INTERVAL_DELIM SYM_LT duration_value SYM_INTERVAL_DELIM
        | SYM_INTERVAL_DELIM SYM_LE duration_value SYM_INTERVAL_DELIM
        | SYM_INTERVAL_DELIM SYM_GT duration_value SYM_INTERVAL_DELIM
        | SYM_INTERVAL_DELIM SYM_GE duration_value SYM_INTERVAL_DELIM
        ;

    term_code: V_QUALIFIED_TERM_CODE_REF
        ;

    term_code_list_value: term_code ’,’ term_code
        | term_code_list_value ’,’ term_code
        | term_code ’,’ SYM_LIST_CONTINUE
        ;
```

2.8.2 Symbols

The following specifies the symbols and lexical patterns used in the above grammar.

```
    ----------/* comments */ -------------------------------------------------
    "--".*                                      -- Ignore comments
    "--".*\n[ \t\r]*

    ----------/* symbols */ -------------------------------------------------
    “-”      Minus_code
    “+”      Plus_code
    “*”      Star_code
    “/”      Slash_code
    “^”      Caret_code
    “.”      Dot_code
    “;”      Semicolon_code
    “,”      Comma_code
    “:”      Colon_code
    “!”      Exclamation_code
    “(“      Left_parenthesis_code
    “)”      Right_parenthesis_code
    “$”      Dollar_code
    “?”      Question_mark_code

    “|”      SYM_INTERVAL_DELIM

    “[“      Left_bracket_code
    “]”      Right_bracket_code

    “=”      SYM_EQ

    “>=”     SYM_GE
    “<=”     SYM_LE

    “<“      SYM_LT / SYM_START_DBLOCK

    “>”      SYM_GT / SYM_END_DBLOCK
```

```
"..”       SYM_ELLIPSIS
"...”      SYM_LIST_CONTINUE

----------/* keywords */ -----------------------------------------------
[Tt][Rr][Uu][Ee]                       SYM_TRUE
[Ff][Aa][Ll][Ss][Ee]                   SYM_FALSE
[Ii][Nn][Ff][Ii][Nn][Ii][Tt][Yy]       SYM_INFINITY
[Qq][Uu][Ee][Rr][Yy]                   SYM_QUERY_FUNC

----------/* term code reference of form [ICD10AM(1998)::F23 */ ----------
\[[a-zA-Z0-9()._\-]+::[a-zA-Z0-9._\-]+\]    V_QUALIFIED_TERM_CODE_REF
\[[a-zA-Z0-9][a-zA-Z0-9._\-]*\]             V_LOCAL_TERM_CODE_REF

----------/* local code definition */ -----------------------------------
a[ct][0-9.]+                                V_LOCAL_CODE

----------/* ISO 8601 duration PNdNNhNNmNNs */ -----------------------------
P([0-9]+[dD])?([0-9]+[hH])?([0-9]+[mM])?([0-9]+[sS])?   V_ISO8601_DURATION

----------/* identifiers */ ---------------------------------------------
[a-zA-Z][a-zA-Z0-9_]*                       V_IDENTIFIER

----------/* numbers */ -----------------------------------------------
[0-9]+                                      V_INTEGER

[0-9]+\./[^.0-9]|
[0-9]+\.[0-9]*[eE][+-]?[0-9]+|
[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?            V_REAL

----------/* Strings */ -----------------------------------------------
\"[^\\\n"]*\"                               V_STRING

---- strings containing quotes, special characters etc
\"[^\\\n"]*                          -- beginning of a string
<IN_STR>\\\\                         -- \\ - append ‘\’
<IN_STR>\\\"                         -- \” - append ‘”’
<IN_STR>&[a-zA-Z][a-zA-Z0-9_]*;      -- match ISO special character
                                     -- pattern &char_name;
<IN_STR>&#x([a-fA-F0-9_]){4};        -- match W3C XML special character
                                     -- pattern &#xHHHH;
<IN_STR>[^\\\n"]+
<IN_STR>\\\n[ \t\r]*                 -- match LF in line
<IN_STR>[^\n"]*\"                    -- match final end of string

<IN_STR>.|\n|                        -- Error
<IN_STR><<EOF>>
```

2.9 Path Syntax

2.9.1 Grammar

This section shows the grammar for paths in both dADL and cADL parts of an archetype, as implemented and tested in the *open*EHR implementation project.

```
--
-- path grammar @changeset 1.24, openEHR implem-dev BK repository
--
```

```
input: og_path
     | error
     ;

og_path: object_path
     | relation_path
     ;

relation_path: attr_path_segment
     | object_path attr_path_segment
     ;

object_path: obj_path_segment
     | relation_path obj_path_segment
     ;

attr_path_segment: V_ATTRIBUTE_IDENTIFIER
     ;

obj_path_segment: V_LOCAL_TERM_CODE_REF '/' -- identified object
     | '/'          -- anonymous object
     ;


call_path: object_path '.' V_ATTRIBUTE_IDENTIFIER
     | call_path '.' V_ATTRIBUTE_IDENTIFIER
     | object_path '.' error
     ;
```

2.9.2 Symbols

The following specifies the symbols and lexical patterns used in the path grammar.

```
----------/* symbols */ -------------------------------------------------
"."         Dot_code
"/"         Slash_code

"["         Left_bracket_code
"]"         Right_bracket_code

----------/* term code reference */ -------------------------------------
\[[a-zA-Z0-9][a-zA-Z0-9._\-]*\]       V_LOCAL_TERM_CODE_REF

----------/* identifiers */ ---------------------------------------------
[A-Z][a-zA-Z0-9_]*                    V_TYPE_IDENTIFIER

[a-z][a-zA-Z0-9_]*                    V_ATTRIBUTE_IDENTIFIER
```

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
          -- etc --
       }
    }
    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
          -- etc --
       }
    }
    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 39.

3.2 Basics

3.2.1 Keywords

The following keywords are recognised in cADL:

- `matches`, `~matches`, `is_in`, `~is_in`
- `occurrences`, `existence`, `cardinality`
- `ordered, unordered`, `unique`
- `infinity`
- `use_node`, `allow_archetype`[1]
- `invariant, include, exclude`

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	$\in$	Set membership, “p is in P”
exists	$\exists$	Existence quantifier, “there exists ...”
for_all	$\forall$	Universal quantifier, “for all x...”
implies	$\supset$	Material implication, “p implies q”, or “if p then q”
and	$\wedge$	Logical conjunction, “p and q”
or	$\vee$	Logical disjunction, “p or q”
xor	$\veebar$	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 the following example, the `matches` operator links the name of an entity to a linear value space (i.e. a list), consisting of all words ending in “ion”.

```
aaa matches {/.*ion[^ ]/}-- the set of english words ending in 'ing'
```

The following example links the name of a type XXX a complex multi-dimensional value space.

1. was ‘use_archetype’, which is now deprecated

```
    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 can be used to constrain 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 for all cADL nodes. Identifiers obey the same rules as in dADL: type names commence with an upper case letter, while attribute and function names commence with a lower case letter. 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.

We use the term "object" block or node to refer to any block introduced by a type name (in this document, in all upper case), while an "attribute" block or node is any block introduced by an attribute identifier (in all lower case in this document), as illustrated below.

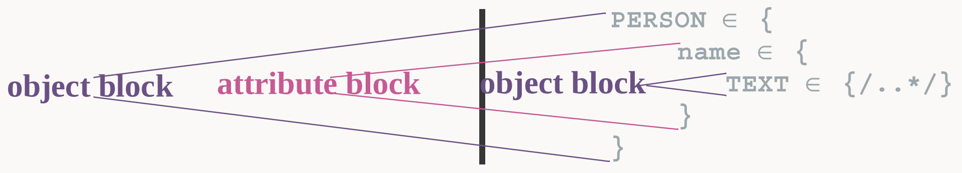


FIGURE 3 Object and Attribute Blocks in cADL

3.3.1 Complex Objects

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.

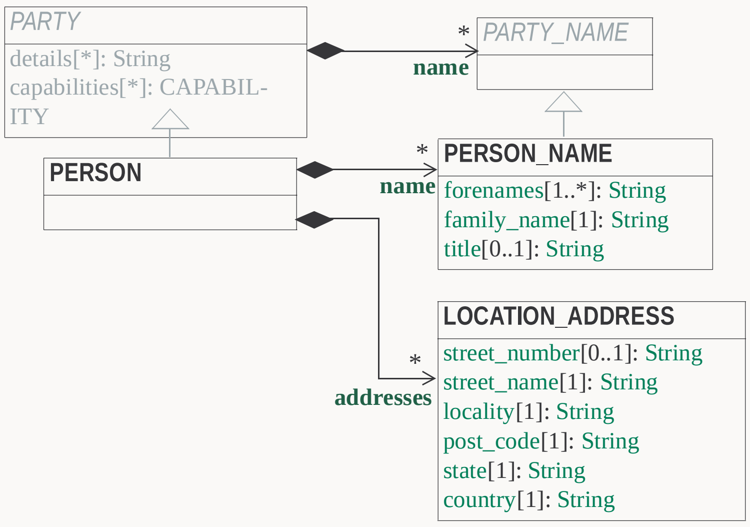


FIGURE 4 UML Model of PERSON

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.

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 Attribute Constraints

Note: in cADL, the term ‘attribute’ denotes any stored property of a type, including primitive attributes and any kind of relationship such as association or aggregation.

In any information model, attributes are either single-valued or multiply-valued, i.e. of a generic container type such as `List<Contact>`. The only constraint that applies to all attributes is to do with existence.

3.3.2.1 Existence

Existence constraints say whether an attribute must exist, and are indicated by “0..1” or “1” markers at line ends in UML diagrams (and often mistakenly referred to as a “cardinality of 1..1”). It is the absence or presence of the cardinality constraint which indicates in cADL that the attribute as being single being constrained is single-valued or a container attribute, respectively. Existence constraints are expressed in cADL 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 mandatory or optional (i.e. obligatory or not) in runtime data for the attribute in question. The above example indicates that a value for the ‘units’ attribute is optional. The same logic applies whether the attribute is of single or multiple cardinality, i.e. whether it is a container or not. For container attributes, the existence constraint indicates whether the whole container (usually a list or set) is mandatory or not; a further *cardinality* constraint (described below) indicates how many members in the container are allowed.

> An **existence constraint** may be used directly after any attribute identifier, and indicates whether the object to which the attribute refers is mandatory or optional in the data.

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}`.

3.3.3 Single-valued Attributes

Repeated blocks of object constraints of the same class (or its subtypes) can have two possible meanings in cADL, depending on whether the cardinality is present or not in the containing attribute block. With no cardinality, the meaning is that each child object constraint of the attribute in question is a possible alternative for the value of the attribute in the date, as shown in the following example:

```
ELEMENT[04] matches {                   -- speed limit
    value matches {
        QUANTITY matches {
            magnitude matches {0..55}
            property matches {"velocity"}
            units matches {"mph"}       -- miles per hour
        }
        QUANTITY 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.

> Two or more "object" blocks introduced by type names appearing after an attribute which is not a container (i.e. for which there is no cardinality constraint) are taken to be **alternative constraints**, only one of which needs to be matched by the data.

Note that there is a more efficient way to express the above example, using domain type extensions. An example is provided in section 6.1.2 on page 81.

3.3.4 Container Attributes

Container attributes are indicated in cADL with the *cardinality* constraint. Cardinalities indicate limits for the number of members of container types such as lists and sets. Consider the following example:

```
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
    }
}
```

3.3.4.1 Cardinality

The keyword cardinality indicates firstly that the property events must be of a container type, such as List<T>, Set<T>, Bag<T>. The integer range indicates the valid membership of the container; a single '*' means the range 0..*, i.e. '0 to many'. The type of the container is not explicitly indicated , since it is usually defined by the information model. However, the semantics of a logical set (unique membership, ordering not significant), a logical list (ordered, non-unique membership) or a bag (unordered, non-unqique membership) can be constrained using the additional keywords ordered, unordered, unique and non-unique within the cardinality constraint, as per the following examples:

```
events cardinality ∈ {*; ordered} ∈ {                  -- logical list
events cardinality ∈ {*; unordered; unique} ∈ {        -- logical set
events cardinality ∈ {*; unordered} ∈ {                -- logical bag
```

In theory, none of these constraints can be stronger than the semantics of the corresponding container in the relevant part of the reference model. However, in practice, developers often use lists to facilitate interation, when the actual semantics are intended to be of a set; in such cases, they typically ensure set-like semantics in their own code rather than by using an Set<T> type. How such constraints are evaluated in practice may depend somewhat on knowledge of the software system.

> A **cardinality constraint** may be used after any attribute name (or after its existence constraint, if there is one) in order to indicate that the attribute refers to a container type, what number of member items it must have in the data, and optionally, whether it has "list" or "set" semantics, via the used of the keywords ordered, unordered and unique.

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..*} ∈ {-- etc --}
```

3.3.4.2 Occurrences

A constraint on occurrences is used only with cADL object nodes (not attribute nodes), to indicate how many times in runtime data an instance of a given class conforming to a particular constraint can occur. It only has significance for objects which are children of a container attribute, since by definition, the occurrences of an object which is the value of a single-valued attribute can only be 0..1 or 1..1, and this is already defined by the attribute occurrences. However, it is not illegal. 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 `GROUP`s 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”}
                -- etc --
            }
            PERSON[105] occurrences ∈ {0..*} matches {
                title ∈ {“member”}
                -- etc --
            }
        }
    }
```

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`, if none is mentioned, is `{1..1}`.

An **occurrences constraint** may appear directly after any type name, in order to indicate how many times data objects conforming to the block introduced by the type name may occur in the data.

3.3.5 “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 {*}
        -- etc --
    }
```

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.

3.3.6 Node Identification

In many of the examples above, some of the object node 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 shown in section 3.3.1, 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 69.

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 [17] 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 [15] 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 ADL node id and the *open*EHR *meaning* attribute, respectively); optionally they can also have a runtime name in the *code.original_text* attribute.

3.3.7 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 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.

3.3.8 Archetype Internal References

It occurs reasonably often that one needs to include a constraint which is essentially a repeat of an earlier complex constraint, but within a different block. This is achieved using an archetype internal reference, according to the following rule:

> An **archetype internal reference** is introduced 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 ∈ {
        -- etc --
    }
    contacts cardinality ∈ {0..*} ∈ {
        CONTACT[002] ∈ {                              -- home address
            purpose ∈ {-- etc --}
            addresses ∈ {-- etc --}
        }
        CONTACT[003] ∈ {                              -- postal address
```

```
            purpose ∈ {-- etc --}
            addresses ∈ {-- etc --}
         }
         CONTACT[004] ∈ {                                 -- home contact
            purpose ∈ {-- etc --}
            addresses cardinality ∈ {0..*} ∈ {
               ADDRESS[005] ∈ {                           -- phone
                  type ∈ {-- etc --}
                  details ∈ {-- etc --}
               ADDRESS[006] ∈ {                           -- fax
                  type ∈ {-- etc --}
                  details ∈ {-- etc --}
               }
               ADDRESS[007] ∈ {                           -- email
                  type ∈ {-- etc --}
                  details ∈ {-- etc --}
               }
            }
         }
         CONTACT[008] ∈ {                                 -- work contact
            purpose ∈ {-- etc --}
            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
            }
         }
      }
```

3.3.9 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. Invariants appear in an invariant block, which is the last part of any object block in an archetype, and are expressed in the archetype assertion language, described in section 3.5 on page 58.

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 object 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
        let $speed_kmh = [001]/speed[002]/kilometres/
        let $speed_mph = = [003]/speed[004]/miles/
        validity: $speed_kmh = $speed_mph * 1.6
```

To Be Continued: the '1.6' above should be coded and included in the ontology section

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 ∈ {
        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
     }
```

This structure follows one possible guideline which is that each invariant should be placed in the innermost object block possible. It may be determined by experience in the future that a single invariant section per archetype is preferable.

3.3.10 Archetype Slots

At any point in a cADL definition, a constraint can be defined which allows other archetypes to be used, rather than defining the desired constraints inline. This is known as an archetype “slot”, i.e. a set of statements which limit what archetypes are allowed to come next, written in the ADL assertion language. The slot might be “wide”, meaning it allows numerous other archetypes, or “narrow”, where it allows only a few or just one archetype. The point at where the slot occurs in the archetype is a “chaining point”, i.e. a point at which archetypes are chained together. This might happen inside a PERSON archetype for example, where an ADDRESS archetype is referred to. The full semantics of archetype chaining are described in the “Archetype Object Model” document. 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.

> An **archetype slot** is introduced with the keyword `allow_archetype`, and is expressed using two lists of cADL invariants, each introduced with the keywords `include` and `exclude`, respectively.

The following example shows how the “Objective” SECTION in a problem/SOAP headings archetype defines two slots, indicating which ENTRY and SECTION archetypes are allowed and excluded under the *items* property.

```
     SECTION[at2000] occurrences ∈ {0..1} ∈ {          -- objective
         items ∈ {
            allow_archetype ENTRY occurrences ∈ {0..1} ∈ {
               include
                  concept_short_name ∈ {/.+/}
            }
            allow_archetype SECTION occurrences ∈ {0..*} ∈ {
               include
```

```
                  id ∈ {/.*\.iso-ehr\.section\..*\..*/}
               exclude
                  id ∈ {/.*\.iso-ehr\.section\.patient_details\..*/}
            }
         }
     }
```

Here, every constraint inside the block starting on an `allow_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:

```
     allow_archetype ENTRY occurrences matches {0..*} matches {
         include
            short_concept_name ∈ {"family_history_subject"}
               and exists /subject/.relation
                  implies /subject/.relation matches {
                     CODED_TEXT matches {
                        code matches {[ac0003]}    -- "parent" or "sibling"
                     }
                  }
     }
```

Note that in the examples above, and in all ADL, the '\' character in paths is not a path separator, as some Microsoft Windows users might assume - it retains its meaning as a quoting character; above it is used to quote the '.' character to ensure it has its literal meaning of '.' rather than "any character" which is its regular expression meaning. Although not particularly beautiful, the syntax used above to indicate allowed values of id is completely standard regular expression syntax, and would be familiar to most users of unix/linux etc operating systems, Perl, and many other formalisms. Developers of archetypes using GUI tools should of course be spared such technical details.

3.3.11 Mixed Structures

Three types of structure which represent constraints on complex objects have been presented so far:

- *complex object structures*: any node introduced by a type name and followed by {} containing constraints on attributes, invariants etc;
- *internal references*: any node introduced by the keyword `use_node`, followed by a type name; such nodes stand for a complex object constraint which has already been expressed elsewhere in the archetype;
- *archetype slots*: any node introduced by the keyword `allow_archetype`, followed by a type name; such nodes stand for a complex object constraint which is expressed in some other archetype.

At any given node, all three types can co-exist, as in the following example:

```
     SECTION[at2000] ∈ {
         items cardinality ∈ {0..*; ordered} ∈ {
            ENTRY[at2001] ∈ {-- etc --}
            allow_archetype ENTRY ∈ {-- etc --}
            use_node ENTRY [001]/some_path[004]/
            ENTRY[at2002] ∈ {-- etc --}
            use_node ENTRY [at1002]/some_path[1012]/
            use_node ENTRY [at1005]/some_path[1052/
            ENTRY[at2003] ∈ {-- etc --}
```

```
        }
    }
```

Here, we have a constraint on an attribute called *items* (of cardinality 0..*), expressed as a series of possible constraints on objects of type `ENTRY`. The 1st, 4th and 7th are described "in place" (the details are removed here, for brevity); the 3rd, 5th and 6th are expressed in terms of internal references to other nodes earlier in the archetype, while the 2nd is an archetype slot, whose constraints are expressed in other archetypes matching the include/exclude constraints appearing between the braces of this node (again, avoided for the sake of brevity). Note also that the `ordered` keyword has been used to indicate that the list order is intended to be significant.

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. All constraints on strings are case-sensitive. 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 69.

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 [18] 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` (case sensitive). 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` (case sensitive). E.g. `/[0-9]/` matches any single character string containing a single decimal digit, /[S-Z]/ matches any single character in the range `S` - `Z`;

`[^a-m]` 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 is 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 lower-case 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 lower-case 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 constrain 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 constrained 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|}       -- allow 0 - 1000
length matches {|<10|}           -- allow up to 9
length matches {|>10|}           -- allow 11 or more
length matches {|<=10|}          -- allow up to 10
length matches {|>=10|}          -- allow 10 or more
length matches {|100+/-5|}       -- allow 100 +/- 5, i.e. 95 - 105
```

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

`infinity`	
`-infinity`	
`*`	`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 28. 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 28, 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"`

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
    magnitude matches {|<10.0|}             -- allow anything less than 10.0
    magnitude matches {|>10.0|}             -- allow greater than 10.0
    magnitude matches {|<=10.0|}            -- allow up to 10.0
    magnitude matches {|>=10.0|}            -- allow 10.0 or more
    magnitude matches {|80.0+/-12.0|}       -- allow 80 +/- 12
```

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

Implied Type	Pattern	Explanation
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 the data correspeond to which values in the list; and if lists are open or closed?

3.5 Assertions

3.5.1 Overview

The assertion part of cADL is a small language 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 [11] languages and tools for over a decade. (See Kilov & Ross [8] and Sowa [14] for an explanation of predicate logic in information modelling.)

3.5.2 Operators

Assertion expressions can include the following operators:

quantifiers: **for_all**, **exists**

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

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

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

The textually named operators among the above all have symbolic equivalents, described at the beginning of this chapter. Parentheses can be used to override standard precedence rules.

To Be Continued: temporal expressions

3.5.3 Operands

Operands in an assertion expression can be any of the following:

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

variable reference: any name starting with '$', e.g. $body_weight;

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 assertion are those referring to nodes and properties inside the block in which the invariant appears.

3.5.4 Variables

Predefined Variables

A number of predefined variables can be referenced in ADL assertion expressions, without prior definition, including

- `$current_date` of type `Date`; returns the date whenever the archetype is evaluated
- `$current_time` of type `Time`; returns time whenever the archetype is evaluated

- `$current_datetime` of type `Date_Time`; returns date/time whenever the archetype is evaluated

To Be Continued:

Archetype-defined Variables

Variables can also be defined inside an archetype, as part of the assertion statements in an invariant. The syntax of variable definition is as follows:

```
let $var_name = reference
```

Here, a reference can be any of the operand types listed above. ‘Let’ statements can come anywhere in an invariant block, but for readability, should generally come first.

The following example illustrates the use of variables in an invariant block:

```
invariant
    let $sys_bp =
       [at0000]/data[at9001]/events[at9002]/data[at1000]/items[at1100]
    let $dia_bp =
       [at0000]/data[at9001]/events[at9002]/data[at1000]/items[at1200]
    $sys_bp >= $dia_bp
```

To Be Continued:

3.6 cADL Syntax

3.6.1 Grammar

This section describes the cADL grammar, as implemented and tested in the *open*EHR implementation project.

```
--
-- cADL grammar @changeset 1.24, openEHR implem-dev BK repository
--

input: c_complex_object
       | error
    ;

c_complex_object: c_complex_object_head SYM_MATCHES SYM_START_CBLOCK
                   c_complex_object_body c_invariants SYM_END_CBLOCK
    ;

c_complex_object_head: c_complex_object_id c_occurrences
    ;

c_complex_object_id: V_TYPE_IDENTIFIER
    | V_TYPE_IDENTIFIER V_LOCAL_TERM_CODE_REF
    ;

c_complex_object_body: c_any
    | c_attributes
    ;

------------------------- node types -----------------------

c_object: c_complex_object
    | archetype_internal_ref
    | archetype_slot
```

```
        | constraint_ref
        | c_coded_term
        | c_ordinal
        | c_primitive_object
        | V_C_DOMAIN_TYPE
        | error
        ;

    archetype_internal_ref: SYM_USE_NODE V_TYPE_IDENTIFIER object_path
        | SYM_USE_NODE V_TYPE_IDENTIFIER error
        ;

    archetype_slot: c_archetype_slot_head SYM_MATCHES SYM_START_CBLOCK
                    c_includes c_excludes SYM_END_CBLOCK
        ;

    c_archetype_slot_head: c_archetype_slot_id c_occurrences
        ;

    c_archetype_slot_id: SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER
        | SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER V_LOCAL_TERM_CODE_REF
        | SYM_ALLOW_ARCHETYPE error
        ;

    c_primitive_object: c_primitive
        ;

    c_primitive: c_integer
        | c_real
        | c_date
        | c_time
        | c_date_time
        | c_duration
        | c_string
        | c_boolean
        | error
        ;

    c_any: '*'
        ;

    ---------------- BODY - relationships ----------------

    c_attributes: c_attribute
        | c_attributes c_attribute
        ;

    c_attribute: c_attr_head SYM_MATCHES SYM_START_CBLOCK c_attr_values
                    SYM_END_CBLOCK
        ;

    c_attr_head: V_ATTRIBUTE_IDENTIFIER c_existence
        | V_ATTRIBUTE_IDENTIFIER c_existence c_cardinality
        ;

    c_attr_values: c_object
        | c_attr_values c_object
        | c_any -- to allow a property to have any value
        | error
        ;

---------------------- invariants ------------------------

c_invariants: -- Empty
        | SYM_INVARIANT invariants
        | SYM_INVARIANT error
        ;

c_includes: -- Empty
        | SYM_INCLUDE invariants
        ;

c_excludes: -- Empty
        | SYM_EXCLUDE invariants
        ;

invariants: invariant
        | invariants invariant
        ;

invariant: any_identifier ‘:’ Boolean_expression
        | Boolean_expression
        | any_identifier ‘:’ error
        ;

---------------------- expressions ---------------------

Boolean_expression: SYM_EXISTS object_path
        | SYM_EXISTS error
        | ‘(‘ Boolean_expression ‘)’
        | call_path -- for boolean returning functions
          -- TEMPORARY: the following only allows single calls,
          -- not multiple dot-form calls on an object;
          -- this can be added fairly easily
        | V_ATTRIBUTE_IDENTIFIER SYM_MATCHES SYM_START_CBLOCK
          c_primitive_object SYM_END_CBLOCK
        | SYM_TRUE
        | SYM_FALSE
        | SYM_NOT Boolean_expression
        | Arithmetic_expression ‘=’ Arithmetic_expression
        | Arithmetic_expression SYM_NE Arithmetic_expression
        | Arithmetic_expression SYM_LT Arithmetic_expression
        | Arithmetic_expression SYM_GT Arithmetic_expression
        | Arithmetic_expression SYM_LE Arithmetic_expression
        | Arithmetic_expression SYM_GE Arithmetic_expression
        | Boolean_expression SYM_AND Boolean_expression
        | Boolean_expression SYM_OR Boolean_expression
        | Boolean_expression SYM_XOR Boolean_expression
        | Boolean_expression SYM_AND SYM_THEN Boolean_expression %prec SYM_AND
        | Boolean_expression SYM_OR SYM_ELSE Boolean_expression %prec SYM_OR
        | Boolean_expression SYM_IMPLIES Boolean_expression
        ;

Arithmetic_expression: call_path
```

```
        | ‘(‘ Arithmetic_expression ‘)’
        | V_INTEGER
        | V_REAL
        | V_STRING
        | V_CHARACTER
        | ‘+’ Arithmetic_expression %prec SYM_NOT
        | ‘-’ Arithmetic_expression %prec SYM_NOT
        | Arithmetic_expression ‘+’ Arithmetic_expression
        | Arithmetic_expression ‘-’ Arithmetic_expression
        | Arithmetic_expression ‘*’ Arithmetic_expression
        | Arithmetic_expression ‘/’ Arithmetic_expression
        | Arithmetic_expression ‘^’ Arithmetic_expression
        | Arithmetic_expression SYM_MODULO Arithmetic_expression
        | Arithmetic_expression SYM_DIV Arithmetic_expression
        ;


---------------- existence, occurrences, cardinality ----------------

c_existence:  -- default to 1..1
        | SYM_EXISTENCE SYM_MATCHES SYM_START_CBLOCK existence_spec
          SYM_END_CBLOCK
        ;

existence_spec:  V_INTEGER -- can only be 0 or 1
        | V_INTEGER SYM_ELLIPSIS V_INTEGER -- can only be 0..0, 0..1, 1..1
        ;

c_cardinality: SYM_CARDINALITY SYM_MATCHES SYM_START_CBLOCK cardinality_spec
               SYM_END_CBLOCK
        ;

cardinality_spec: occurrence_spec
        | occurrence_spec ‘;’ SYM_ORDERED
        | occurrence_spec ‘;’ SYM_UNORDERED
        | occurrence_spec ‘;’ SYM_UNIQUE
        | occurrence_spec ‘;’ SYM_ORDERED ‘;’ SYM_UNIQUE
        | occurrence_spec ‘;’ SYM_UNORDERED ‘;’ SYM_UNIQUE
        | occurrence_spec ‘;’ SYM_UNIQUE ‘;’ SYM_ORDERED
        | occurrence_spec ‘;’ SYM_UNIQUE ‘;’ SYM_UNORDERED
        ;

cardinality_limit_value: integer_value
        | ‘*’
        ;

c_occurrences:  -- default to 1..1
        | SYM_OCCURRENCES SYM_MATCHES SYM_START_CBLOCK occurrence_spec
          SYM_END_CBLOCK
        | SYM_OCCURRENCES error
        ;

occurrence_spec: cardinality_limit_value -- single integer or ‘*’
        | V_INTEGER SYM_ELLIPSIS cardinality_limit_value
        ;

---------------------- leaf constraint types -----------------------
```

```
    c_ordinal: ordinal
        | c_ordinal ‘,’ ordinal
        ;

    ordinal: integer_value SYM_INTERVAL_DELIM V_TERM_CODE_CONSTRAINT
        ;

    c_integer: integer_value
        | integer_list_value
        | integer_interval_value
        | occurrence_spec
        ;

    c_real: real_value
        | real_list_value
        | real_interval_value
        ;

    c_date: V_ISO8601_DATE_CONSTRAINT_PATTERN
        | date_value
        | date_interval_value
        ;

    c_time: V_ISO8601_TIME_CONSTRAINT_PATTERN
        | time_value
        | time_interval_value
        ;

    c_date_time: V_ISO8601_DATE_TIME_CONSTRAINT_PATTERN
        | date_time_value
        | date_time_interval_value
        ;

    c_duration: duration_value
        | duration_interval_value
        ;

    c_string:  V_STRING -- single value, generates closed list
        | string_list_value -- closed list
        | string_list_value ‘,’ SYM_LIST_CONTINUE -- open list
        | V_REGEXP -- regular expression with “//” or “^^” delimiters
        ;

    c_boolean: SYM_TRUE
        | SYM_FALSE
        | SYM_TRUE ‘,’ SYM_FALSE
        | SYM_FALSE ‘,’ SYM_TRUE
        ;

    c_coded_term: V_TERM_CODE_CONSTRAINT -- e.g. “local::at0040”
        ;

    constraint_ref: V_LOCAL_TERM_CODE_REF -- e.g. “ac0003”
        ;

    any_identifier: V_TYPE_IDENTIFIER
```

```
        | V_ATTRIBUTE_IDENTIFIER
        ;
```

3.6.2 Symbols

The following shows the lexical specification for the cADL grammar.

```
----------/* comments */ -------------------------------------------------
"--".*                                      -- Ignore comments
"--".*\n[ \t\r]*

----------/* symbols */ -------------------------------------------------
"-"        Minus_code
"+"        Plus_code
"*"        Star_code
"/"        Slash_code
"^"        Caret_code
"="        Equal_code
"."        Dot_code
";"        Semicolon_code
","        Comma_code
":"        Colon_code
"!"        Exclamation_code
"("        Left_parenthesis_code
")"        Right_parenthesis_code
"$"        Dollar_code
"?"        Question_mark_code

"|"        SYM_INTERVAL_DELIM

"["        Left_bracket_code
"]"        Right_bracket_code

"{"        SYM_START_CBLOCK
"}"        SYM_END_CBLOCK

">="       SYM_GE
"<="       SYM_LE
"!="       SYM_NE

"<"        SYM_LT
">"        SYM_GT

"\\"       SYM_MODULO
"//"       SYM_DIV

".."       SYM_ELLIPSIS
"..."      SYM_LIST_CONTINUE

----------/* keywords */ -------------------------------------------------

[Tt][Hh][Ee][Nn]                                 SYM_THEN

[Ee][Ll][Ss][Ee]                                 SYM_ELSE

[Aa][Nn][Dd]                                     SYM_AND
```

```
[Oo][Rr]                                          SYM_OR

[Xx][Oo][Rr]                                      SYM_XOR

[Nn][Oo][Tt]                                      SYM_NOT

[Ii][Mm][Pp][Ll][Ii][Ee][Ss]                      SYM_IMPLIES

[Tt][Rr][Uu][Ee]                                  SYM_TRUE

[Ff][Aa][Ll][Ss][Ee]                              SYM_FALSE


[Ff][Oo][Rr][_][Aa][Ll][Ll]                       SYM_FORALL

[Ee][Xx][Ii][Ss][Tt][Ss]                          SYM_EXISTS

[Ee][Xx][Ii][Ss][Tt][Ee][Nn][Cc][Ee]              SYM_EXISTENCE

[Oo][Cc][Cc][Uu][Rr][Rr][Ee][Nn][Cc][Ee][Ss]      SYM_OCCURRENCES

[Cc][Aa][Rr][Dd][Ii][Nn][Aa][Ll][Ii][Tt][Yy]      SYM_CARDINALITY

[Oo][Rr][Dd][Ee][Rr][Ee][Dd]                      SYM_ORDERED

[Uu][Nn][Oo][Rr][Dd][Ee][Rr][Ee][Dd]              SYM_UNORDERED

[Uu][Nn][Ii][Qq][Uu][Ee]                          SYM_UNIQUE

[Mm][Aa][Tt][Cc][Hh][Ee][Ss]                      SYM_MATCHES

[Ii][Ss]_[Ii][Nn]                                 SYM_MATCHES

[Ii][Nn][Vv][Aa][Rr][Ii][Aa][Nn][Tt]              SYM_INVARIANT

[Ii][Nn][Ff][Ii][Nn][Ii][Tt][Yy]                  SYM_INFINITY

[Uu][Ss][Ee][_][Nn][Oo][Dd][Ee]                   SYM_USE_NODE

[Uu][Ss][Ee][_][Aa][Rr][Cc][Hh][Ee][Tt][Yy][Pp][Ee]  SYM_ALLOW_ARCHETYPE

[Aa][Ll][Ll][Oo][Ww][_][Aa][Rr][Cc][Hh][Ee][Tt][Yy][Pp][Ee]
                                    SYM_ALLOW_ARCHETYPE -- deprecated

[Ii][Nn][Cc][Ll][Uu][Dd][Ee]SYM_INCLUDE

[Ee][Xx][Cc][Ll][Uu][Dd][Ee]SYM_EXCLUDE


---------/* qualified term code reference */ ------------------------
-- any qualified code, e.g. [local::at0001], [local::ac0001], [loinc::700-0]
--
\[[a-zA-Z0-9()._\-]+::[a-zA-Z0-9._\-]+\]          V_TERM_CODE_CONSTRAINT

---------/* term code constraint pattern of form:  */ ------------
-- [terminology_id::code, -- comment
--              code, -- comment
--            code] -- comment
{mini parser specification}                           V_TERM_CODE_CONSTRAINT

---------/* local term code reference */ ------------------------------
-- any unqualified code, e.g. [at0001], [ac0001], [700-0]
--
\[[a-zA-Z0-9][a-zA-Z0-9._\-]*\]                       V_LOCAL_TERM_CODE_REF

----------/* local term code definition */ ----------------------------
a[ct][0-9.]+                                          V_LOCAL_CODE

----------/* ISO 8601 duration PNdNNhNNmNNs */ -------------------------
P([0-9]+[dD])?([0-9]+[hH])?([0-9]+[mM])?([0-9]+[sS])?   V_ISO8601_DURATION

----------/* ISO 8601-based date constraint pattern */ -----------------
[yY][yY][yY][yY]-[mM?X][mM?X]-[dD?X][dD?X]
                                     V_ISO8601_DATE_CONSTRAINT_PATTERN

----------/* ISO 8601-based time constraint pattern */ ------------------
[hH][hH]:[mM?X][mM?X]:[sS?X][sS?X]    V_ISO8601_TIME_CONSTRAINT_PATTERN

----------/* ISO 8601-based date/time constraint pattern */ -------------
[yY][yY][yY][yY]-[mM?][mM?]-
[dD?X][dD?X][\t][hH?X][hH?X]:[mM?X][mM?X]:[sS?X][sS?X]
                                     V_ISO8601_DATE_TIME_CONSTRAINT_PATTERN

----------/* identifiers */ ------------------------------------------
[A-Z][a-zA-Z0-9_]*                    V_TYPE_IDENTIFIER
[a-z][a-zA-Z0-9_]*                    V_ATTRIBUTE_IDENTIFIER


----------/* domain types - sections of dADL syntax */ ------------------
{mini-parser specification}           V_C_DOMAIN_TYPE
"{<"                                  -- in dADL block
<IN_C_DOMAIN_TYPE>[^}>]*}             -- match non >} stuff in dADL block
<IN_C_DOMAIN_TYPE>[^}>]*>[ \n]*[^}]   -- match to next > not followed by a }
<IN_C_DOMAIN_TYPE>[^}>]*>+[ \n]*}     -- final section

----------/* regular expressions */ -------------------------------------
{mini-parser specification}           V_REGEXP
"{/"                                  -- start of regexp
<IN_REGEXP1>[^/]*\\\/        -- match any segments with quoted slashes
<IN_REGEXP1>[^/}]*\/         -- match final segment

\^[^^\n]*\^{                 -- regexp formed using '^' delimiters

----------/* numbers */ -------------------------------------------------
[0-9]+                                      V_INTEGER

[0-9]+\./[^.0-9]|
[0-9]+\.[0-9]*[eE][+-]?[0-9]+|
[0-9]*\.[0-9]+([eE][+-]?[0-9]+)?            V_REAL

----------/* Strings */ -------------------------------------------------
\"[^\\\n"]*\"                               V_STRING

-- strings containing quotes, special characters etc
```

```
    {mini-parser specification}          V_STRING
    \"[^\\\n"]*                          -- beginning of a string
    <IN_STR>\\\\                         -- \\ - append '\'
    <IN_STR>\\\"                         -- \" - append '"'
    <IN_STR>&[a-zA-Z][a-zA-Z0-9_]*;      -- match ISO special character
                                         -- pattern &char_name;
    <IN_STR>&#x([a-fA-F0-9_]){4};        -- match W3C XML special character
                                         -- pattern &#xHHHH;
    <IN_STR>[^\\\n"]+
    <IN_STR>\\\n[ \t\r]*                 -- match LF in line
    <IN_STR>[^\n"]*\"                    -- match final end of string

    <IN_STR>.|\n|                        -- Error
```

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 81, 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
revision_history
    dADL 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 (adl_version=1.2)
    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 15 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 (adl_version=1.2)
    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 15 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 Language Section

The `language` section includes data describing the original language in which the archetype was authored (essential for evaluating natural language quality), and the total list of languages available in the archetype. There can be only one original_language. The languages_available list should be updated every time a translation of any part of the `description` section, or the `term_definition` and `constraint_definition` subsections in the `ontology` section are translated, and it must include the original_language. The following shows a typical example.

```
language
    original_language = <"en">
    languages_available = <"en", "de">
```

4.2.5 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
    original_author = <"Dr J Joyce">
    original_author_organisation = <"NT Health Service">
    lifecycle_state = <"initial">

    original_resource_uri =
       <"www.healthdata.org.au/data_sets/diabetic_review_data_set_1.html">
    archetype_package_uri =
       <"www.aihw.org.au/data_sets/diabetic_archetypes.html">

    details = <
       [“en”] = <
          purpose = <"archetype for diabetic patient review">
          use = <"used for all hospital or clinic-based diabetic reviews,
             including first time. Optional sections are removed according
             to the particular review">
          misuse = <"not appropriate for pre-diagnosis use">
          other_details = <...>
       >
       [“de”] = <
          purpose = <"archetype for diabetic patient review">
          use = <"used for all hospital or clinic-based diabetic reviews,
             including first time. Optional sections are removed according
             to the particular review">
          misuse = <"not appropriate for pre-diagnosis use">
          other_details = <...>
       >
    >
```

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 details section and its subsections. 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 [16] 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 and other ontological definitions (such as quantitative definitions). Linguistic language translations are added in the form of extra blocks keyed by the relevant language. The following example shows the general layout of this section.

```
ontology
    terminologies_available = <"snomed_ct", ...>

    term_definitions = <
        ["en"] = <...>
        ["de"] = <
            description = <translation = <"acme_translation@acme.com.de">
            ...
        >
    >

    term_binding = <
        ["snomed_ct"] = <...>
        ...
    >

    constraint_definitions = <
        ["en"] = <...>
        ["de"] = <
            description = <translation = <"acme_translation@acme.com.de">
            ...
        >
        ...
    >

    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 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 = <
            [“at0000”] = <
                text = <"problem">
                description = <"The problem experienced by the subject
                    of care to which the contained information relates">
            >
            [“at0001”] = <
                text = <"problem/SOAP headings">
                description = <"SOAP heading structure for multiple problems">
            >
            ...
            [“at4000”] = <
                text = <"plan">
                description = <"The clinician's professional advice">
            >
        >
    >
    [“de”] = <
        description = <
            translation = <
                provenance = <"mdarlison@chimera.upstairs.uk">
                quality_control = <"British Medical Translator’s id 00400595”>
            >
        >
        items = <
            [“at0000”] = <
                text = <"klinisches Problem">
                description = <"Das Problem des Patienten worauf sich diese \
                            Informationen beziehen">
            >
            [“at0001”] = <
                text = <"Problem/SOAP Schema">
                description = <"SOAP-Schlagwort-Gruppierungsschema fuer
                        mehrfache Probleme">
            >
            [“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") = <
         ["umls"] = <
            items =<
               ["at0000"] = <[umls::C124305]> -- apgar result
               ["at0002"] = <[umls::0000000]> -- 1-minute event
               ["at0004"] = <[umls::C234305]> -- cardiac score
               ["at0005"] = <[umls::C232405]> -- respiratory score
               ["at0006"] = <[umls::C254305]> -- muscle tone score
               ["at0007"] = <[umls::C987305]> -- reflex response score
               ["at0008"] = <[umls::C189305]> -- color score
               ["at0009"] = <[umls::C187305]> -- apgar score
               ["at0010"] = <[umls::C325305]> -- 2-minute apgar
               ["at0011"] = <[umls::C725354]> -- 5-minute apgar
```

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

4.5 Revision History Section

The revision history section of an archetype shows the audit history of changes to the archetype, and is expressed in dADL syntax. It is included at the end of the archetype, since it does not contain content of direct interest to archetype authors, and will monotonically grow in size. Where archetypes are stored in a version-controlled repository such as CVS or some commercial product, the revision history section would normally be regenerated each time by the authoring software, e.g. via processing of the output of the 'prs' command used with SCCS files, or 'rlog' for RCS files. The following shows a typical example, with entries in most-recent-first order (although technically speaking, the order is irrelevant to ADL).

```
    revision_history
        revision_history = <
```

```
            ["1.2"] = <
               committer = <"Miriam Hanoosh">
               committer_organisation = <"AIHW.org.au">
               time_committed = <2004-11-02 09:31:04+1000>
               revision = <"1.2">
               reason = <"Added social history section">
               change_type = <"Modification">
            >
            ["1.1"] = <
               committer = <"Enrico Barrios">
               committer_organisation = <"AIHW.org.au">
               time_committed = <2004-09-24 11:57:00+1000>
               revision = <"1.1">
               reason = <"Updated HbA1C test result reference">
               change_type = <"Modification">
            >
            ["1.0"] = <
               committer = <"Enrico Barrios">
               committer_organisation = <"AIHW.org.au">
               time_committed = <2004-09-14 16:05:00+1000>
               revision = <"1.0">
               reason = <"Initial Writing">
               change_type = <"Creation">
            >
         >
```

4.6 Archetype Syntax

4.6.1 Grammar

This section describes the ADL grammar, as implemented and tested in the *open*EHR implementation project.

```
--
-- ADL grammar @changeset 1.24, openEHR implem-dev BK repository
--
input: archetype
    | error
    ;

archetype: arch_identification arch_specialisation arch_concept
           arch_description arch_definition arch_ontology
    ;

arch_identification: SYM_ARCHETYPE V_ARCHETYPE_ID
    | SYM_ARCHETYPE error
    ;

arch_specialisation: -- empty is ok
    | SYM_SPECIALIZE V_ARCHETYPE_ID
    | SYM_SPECIALIZE error
    ;

arch_concept: SYM_CONCEPT V_LOCAL_TERM_CODE_REF
    | SYM_CONCEPT error
    ;
    arch_description: -- no meta-data ok
        | SYM_DESCRIPTION V_DADL_TEXT
        | SYM_DESCRIPTION error
        ;


    arch_definition:SYM_DEFINITION V_CADL_TEXT
        | SYM_DEFINITION error
        ;

    arch_ontology:SYM_ONTOLOGY V_DADL_TEXT
        | SYM_ONTOLOGY error
        ;
```

4.6.2 Symbols

The following shows the ADL lexical specification.

```
    ----------/* symbols */ -------------------------------------------------
    "-"        Minus_code
    "+"        Plus_code
    "*"        Star_code
    "/"        Slash_code
    "^"        Caret_code
    "="        Equal_code
    "."        Dot_code
    ";"        Semicolon_code
    ","        Comma_code
    ":"        Colon_code
    "!"        Exclamation_code
    "("        Left_parenthesis_code
    ")"        Right_parenthesis_code
    "$"        Dollar_code
    "?"        Question_mark_code

    "["        Left_bracket_code
    "]"        Right_bracket_code

    ----------/* keywords */ -------------------------------------------------
    ^[Aa][Rr][Cc][Hh][Ee][Tt][Yy][Pp][Ee][ \t\r]*\n              SYM_ARCHETYPE
    ^[Ss][Pp][Ee][Cc][Ii][Aa][Ll][Ii][SsZz][Ee][ \t\r]*\n        SYM_SPECIALIZE
    ^[Cc][Oo][Nn][Cc][Ee][Pp][Tt][ \t\r]*\n                      SYM_CONCEPT
    ^[Dd][Ee][Ff][Ii][Nn][Ii][Tt][Ii][Oo][Nn][ \t\r]*\n          SYM_DEFINITION
    ^[Dd][Ee][Ss][Cc][Rr][Ii][Pp][Tt][Ii][Oo][Nn][ \t\r]*\n      SYM_DESCRIPTION
    ^[Oo][Nn][Tt][Oo][Ll][Oo][Gg][Yy][ \t\r]*\nSYM_ONTOLOGY

    ----------/* term code reference */ --------------------------------------
    \[[a-zA-Z0-9][a-zA-Z0-9.-]*\]                        V_LOCAL_TERM_CODE_REF

    ----------/* archetype id */ ---------------------------------------------
    [a-zA-Z][a-zA-Z0-9_-]+\.[a-zA-Z][a-zA-Z0-9_-]+\.[a-zA-Z0-9]+V_ARCHETYPE_ID

    ----------/* identifiers */ ----------------------------------------------
    [a-zA-Z][a-zA-Z0-9_]*V_IDENTIFIER
```

5 The ADL Parsing Process

5.1 Overview

FIGURE 5 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.

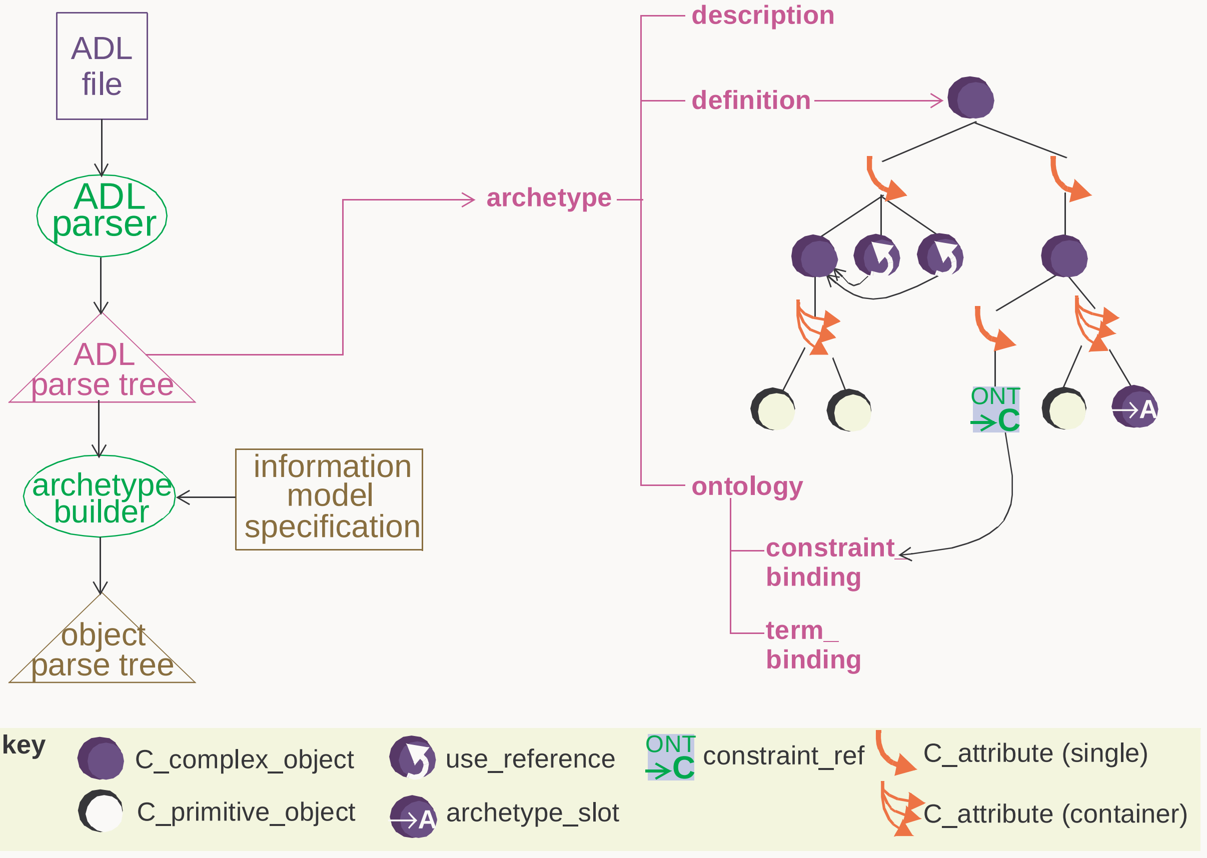


FIGURE 5 Parsed ADL Structure

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:

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

C_attribute: a node representing a constraint on an attribute (i.e. UML 'relationship' or 'primitive attribute') in an object type;

C_primitive_object: an node representing a constraint on a primitive (built-in) object type;

Archetype_internal_ref: a node that refers to a previously defined object node in the same archetype. The reference is made using a path;

Constraint_ref: a node that refers to a constraint on (usually) a text or coded term entity, which appears in the ontology section of the archetype, and in ADL, is referred to with an "acNNNN" code. The constraint is expressed in terms of a query on an external entity, usually a terminology or ontology;

Archetype_slot: a node whose statements define a constraint that determines which other archetypes can appear at that point in the current archetype. It can be thought of like a keyhole, into which few or many keys might fit, depending on how specific its shape is. Logically it has the same semantics as a `C_COMPLEX_OBJECT`, except that the constraints are expressed in another archetype, not the current one.

To Be Continued:

6 Predefined Type Libraries

6.1 Introduction

Standard ADL has a completely regular way of representing constraints. Type names and attribute names from a reference model are mentioned in an alternating, hierarchical structure which is isomorphic to the structure of the corresponding classes in the reference model; constraints at the leaf nodes are represented in a syntactic way which avoids committing to particular modelling details. The overall result enables constraints on most reference model types to be expressed.

6.1.1 Custom Syntax

However, there are occasions for which the standard approach is not enough. One situation is where not everyone in the archetype user base wants to use exactly the same reference model, but nevertheless agrees on the general semantics for many of the types. A typical situation is the type CODE_PHRASE in the *open*EHR reference model, Data_types package. This type models the notion of a 'coded term', which is ubiquitous in clinical computing. Various user communities in health informatics have slightly different models of the 'coded term' concept, yet all would like to share archetypes which constrain it. One way around this is to provide additional syntax enabling such constraints to be expressed, while avoiding mentioning any type or attribute names. The following figure shows how this is done, using the example of CODE_PHRASE.

standard ADL
using type and
attribute names

```
code matches {
    CODE_PHRASE matches {
       terminology_id matches {"local"}
       code_string matches {"at039"} -- lying
    }
    CODE_PHRASE matches {
       terminology_id matches {"local"}
       code_string matches {"at040"} -- sitting
    }
}
```

clinical ADL
syntax

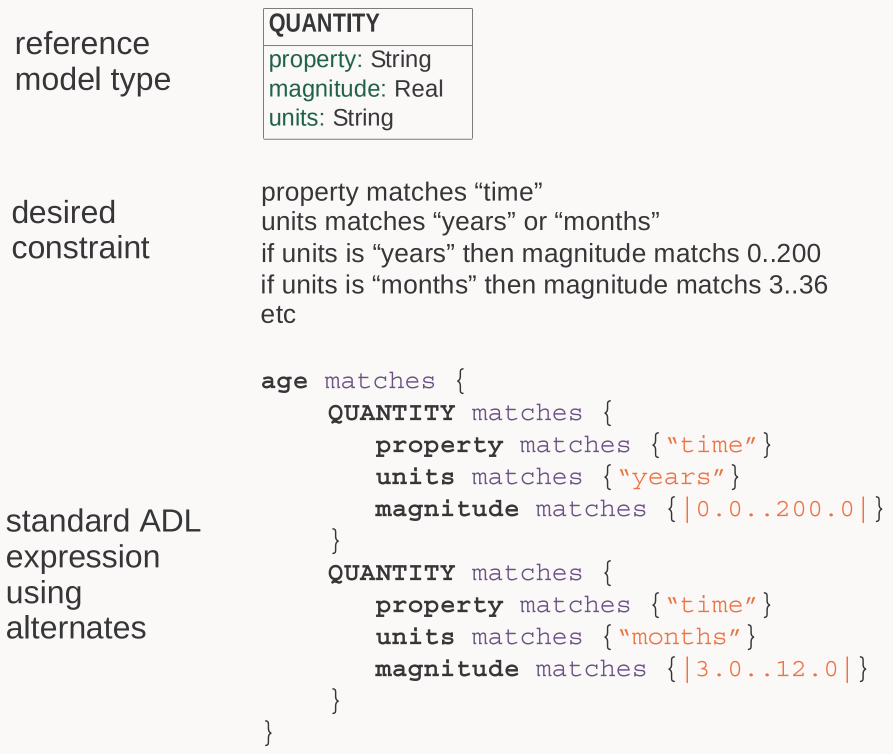
```
code matches {
    [local::
       at039, -- lying
       at040] -- sitting
}
```

FIGURE 6 Constraints using additional syntax

While these two ADL fragments express exactly the same constraint, the second is clearly shorter and clearer, and avoids implying anything about the formal model of the type of the *code* attribute being constrained.

6.1.2 Custom Constraint Classes

Another situation in which standard ADL falls short is when the required semantics of constraint are different from what is provided by the standard approach. Consider a simple type QUANTITY, shown at the top of FIGURE 7, which could be used to represent a person age in data. A typical ADL constraint to enable QUANTITY to be used to represent age in clinical data is shown below, followed by its expression in ADL. The only way to do this in ADL is to use multiple alternatives. While this is a perfectly legal approach, it makes processing by software difficult, since the way such a constraint would be displayed in a GUI would be factored differently.



property matches “time”
units matches “years” or “months”
if units is “years” then magnitude matchs 0..200
if units is “months” then magnitude matchs 3..36
etc

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

FIGURE 7 Standard ADL for Constraint on Quantity

A more powerful possibility is to introduce a new class into the archetype model, representing the concept “constraint on QUANTITY”, which we will call C_QUANTITY here. Such a class fits into the class model of archetypes (described in the *open*EHR Archetype Model document), inheriting from the class C_DOMAIN_TYPE. The C_QUANTITY class is illustrated in FIGURE 10, and corresponds to the way constraints on QUANTITY objects are expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

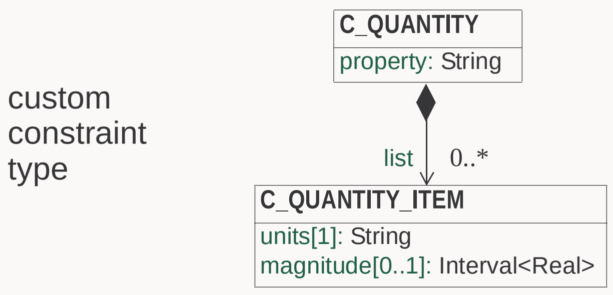


FIGURE 8 Custom Constraint Type for QUANTITY

The question now is how to express a constraint corresponding to this class in an ADL archetype. The solution is logical, and uses standard ADL. Consider that a particular constraint on a QUANTITY must be *an instance* of a `C_QUANTITY`; which can be expressed the appropriate point in the archetype in the form of a section of dADL - the data syntax used in the archetype ontology.

```
value matches {
    C_QUANTITY <
       property = <"time">
       list = <
          items(1) = <
             units = <"yr">
             magnitude = <|0.0..200.0|>
          >
          items(2) = <
             units = <"mth">
             magnitude = <|1.0..36.0|>
          >
       >
    >
}
```

FIGURE 9 Inclusion of a Constraint Object as Data

This approach can be used for any custom type which represents a constraint on a reference model type. The rules are as follows:

- the dADL section occurs inside the {} block where its standard ADL equivalent would have occurred (i.e. no other delimiters or special marks are needed);
- the dADL section must be 'typed', i.e. it must start with a type name, which should be a rule-based transform of a reference model type (as described in Adding Type Information on page 24);
- the dADL instance must obey the semantics of the custom type of which it is an instance.

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

7 Clinical ADL Type Library

7.1 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.

7.2 Custom Syntax Additions

7.2.1 Terminological Types

FIGURE 10 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.

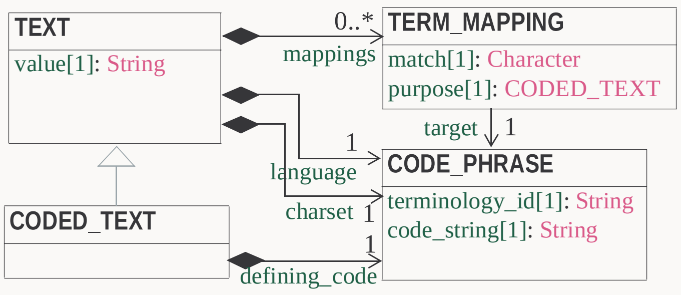


FIGURE 10 TEXT Types

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.

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}/}
```

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 {
          ...
       }
    }
}
```

CODE_PHRASE

7.2.1.1 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
```

7.2.1.2 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.

7.2.2 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.

7.3 Quantitative Types

FIGURE 11 illustrates a partial model of a set of common quantitative types used in science and clinical medicine.

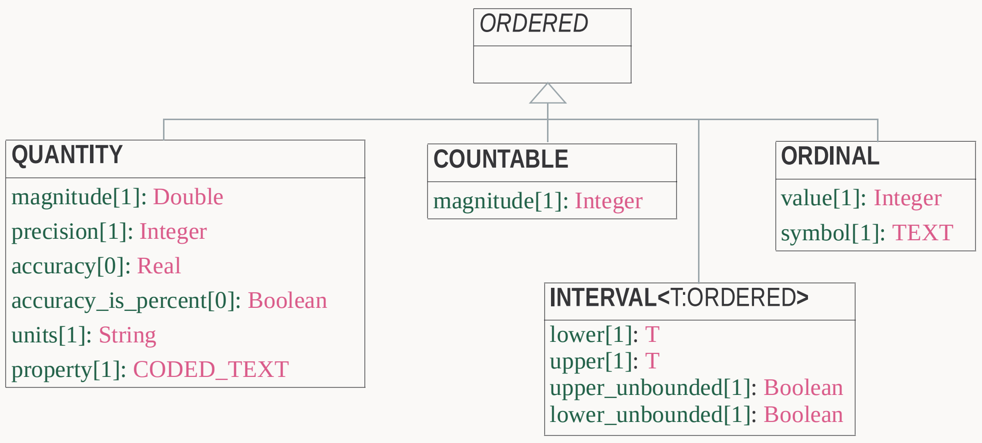


FIGURE 11 Quantitative Types

7.3.1 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.

7.3.2 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}
        }
    }
```

7.3.3 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.

7.3.4 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:

7.4 Date/Time Types

FIGURE 12 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 56); whenever an "XX" or "??" is encountered in a constraint pattern, one of the partial types can be inferred.

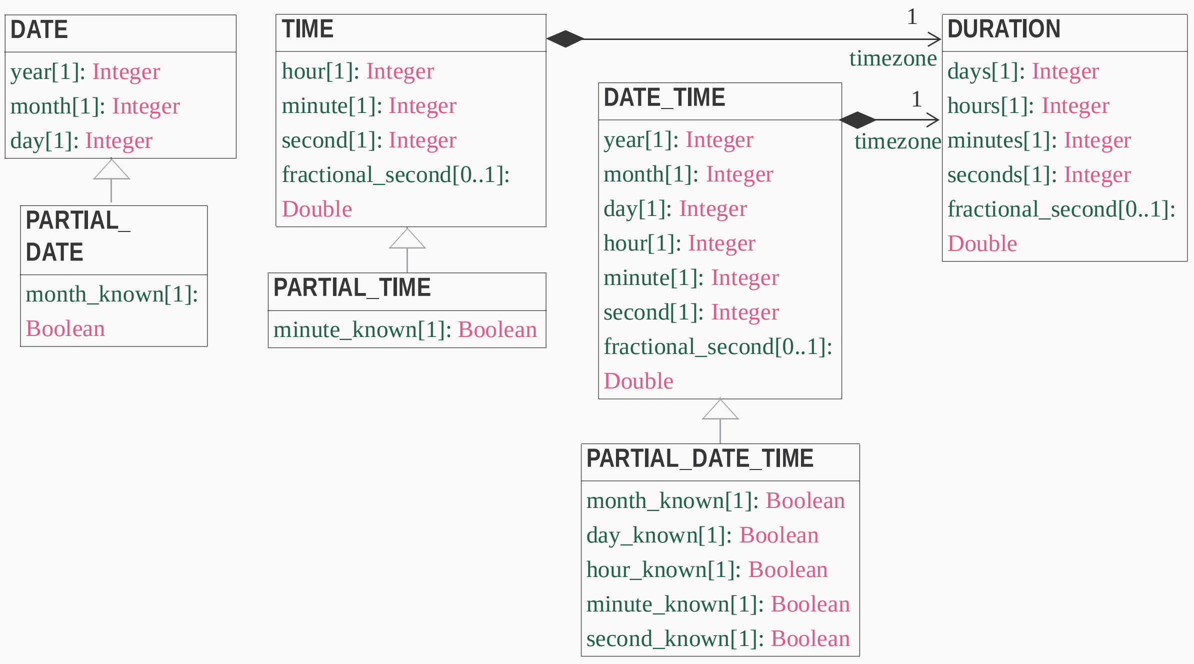


FIGURE 12 Date/Time Types

A References

Publications

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

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

3 Beale T, Heard S. The openEHR Archetype Object Model. See http://www.openehr.org/repositories/spec-dev/latest/publishing/architecture/archetypes/archetype_model/REV_HIST.html.

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

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

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

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

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

9 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).

10 Martin P. Translations between UML, OWL, KIF and the WebKB-2 languages (For-Taxonomy, Frame-CG, Formalized English). May/June 2003. Available at http://meganesia.int.gu.edu.au/~phmartin/WebKB/doc/model/comparisons.html as at Aug 2004.

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

12 Patel-Schneider P, Horrocks I, Hayes P. *OWL Web Ontology Language Semantics and Abstract Syntax*. See http://w3c.org/TR/owl-semantics/.

13 Smith G. *The Object Z Specification Language*. Kluwer Academic Publishers 2000. See http://www.itee.uq.edu.au/~smith/objectz.html.

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

Resources

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

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

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

18 Perl Regular Expressions. See http://www.perldoc.com/perl5.6/pod/perlre.html.

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

20 W3C. *OWL - the Web Ontology Language*.
See http://www.w3.org/TR/2003/CR-owl-ref-20030818/.

21 W3C. XML Path Language. See http://w3c.org/TR/xpath.

END OF DOCUMENT