Release 1.1

The *open*EHR Archetype Model

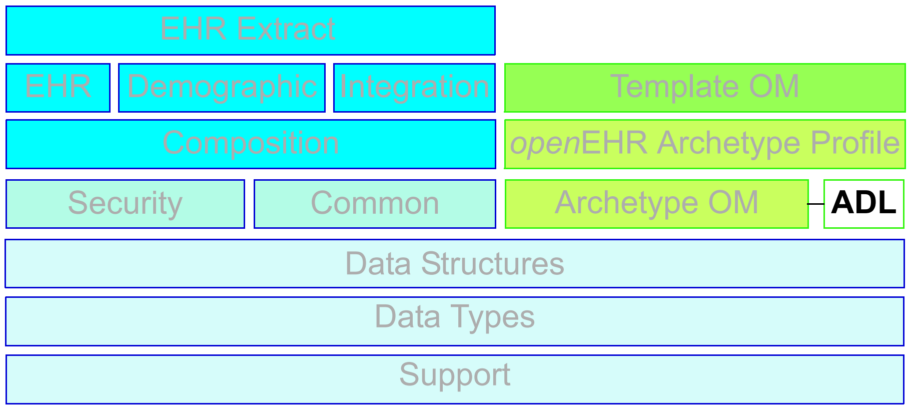
Archetype Definition Language

ADL 1.5

Editors: {T Beale, S Heard}[a]		
Revision: 1.5.0	*Pages:* 140	*Date of issue:* 22 Mar 2009
Status: TRIAL		

a. Ocean Informatics

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



Copyright Notice

Amendment Record

Issue	Details	Raiser	Completed
RELEASE 1.1 candidate			
1.5.0	**SPEC-270**: Add specialisation semantics to ADL and AOM. Improve explanation of node identifiers. Correct typographical errors. **SPEC-xxx**. Add fractional seconds to dADL grammar. **SPEC-xxx**. Remove default existence of 1..1. **SPEC-XXX**. Ensure slot constraints on ARCHETYPE_IDs are full regexes. Added C_STRING.*is_pattern*.	T Beale S Heard O Pishev S Garde S Heard A Flinton	22 Mar 2009
RELEASE 1.0.2			
1.4.1	**SPEC-268**: Correct missing parentheses in dADL type identifiers. dADL grammar and cADL scanner rules updated. **SPEC-284**: Correct inconsistencies in naming of *term_definitions*, *constraint_definitions*, *term_bindings*, *constraint_bindings* attributes in XML-schema and specifications. Improved explanatory text for composite identifiers, including statement on case-sensitivity. Warning on .v1draft non-conformance **SPEC-260**: Correct the regex published for the ARCHETYPE_ID type. Update ADL grammar ARCHEYTPE_ID definition.	R Chen A Torrisi P Gummer, J Arnett, E Browne	12 Dec 2008
RELEASE 1.0.1			
1.4.0	**CR-000203**: Release 1.0 explanatory text improvements. Improve Archetype slot explanation. **CR-000208**: Improve ADL grammar for assertion expressions. **CR-000160**: Duration constraints. Added ISO 8601 patterns for duration in cADL. **CR-000213**: Correct ADL grammar for date/times to be properly ISO8601-compliant. Include 'T' in cADL patterns and dADL and cADL Date/time, Time and Duration values. **CR-000216**: Allow mixture of W, D etc in ISO8601 Duration (deviation from standard). **CR-000200**: Correct Release 1.0 typographical errors. **CR-000225**: Allow generic type names in ADL. **CR-000226**: Rename C_CODED_TEXT to C_CODE_PHRASE **CR-000233**: Define semantics for *occurrences* on ARCHETYPE_INTERNAL_REF. **CR-000241**: Correct cADL grammar for archeype slot match expressions **CR-000223**: Clarify quoting rules in ADL **CR-000242**: Allow non-inclusive two-sided ranges in ADL. **CR-000245**: Allow term bindings to paths in archetypes.	T Beale T Beale S Heard T Beale S Heard A Patterson R Chen S Garde T Beale M Forss T Beale K Atalag S Heard A Patterson S Heard S Heard	13 Mar 2007
RELEASE 1.0			

Issue	Details	Raiser	Completed
1.3.1	**CR-000136**. Add validity rules to ADL document. **CR-000171**. Add validity check for cardinality & occurrences	T Beale A Maldondo	18 Jan 2006
1.3.0	**CR-000141**. Allow point intervals in ADL. Updated atomic types part of cADL section and dADL grammar section. **CR-000142**. Update dADL grammar to support assumed values. **CR-000143**. Add partial date/time values to dADL syntax. **CR-000149**. Add URIs to dADL and remove query() syntax. **CR-000153**. Synchronise ADL and AOM for language attributes **CR-000156**. Update documentation of container types. **CR-000138**. Archetype-level assertions.	S Heard T Beale T Beale T Beale T Beale T Beale T Beale	18 Jun 2005
RELEASE 0.95			
1.2.1	**CR-000125**. C_QUANTITY example in ADL manual uses old dADL syntax. **CR-000115**. Correct "/[xxx]" path grammar error in ADL. Create new section describing ADL path syntax. **CR-000127**. Restructure archetype specifications. Remove clinical constraint types section of document.	T Beale T Beale T Beale	11 Feb 2005
1.2	**CR-000110**. Update ADL document and create AOM document. Added explanatory material; added domain type support; rewrote of most dADL sections. Added section on assumed values, “controlled” flag, nested container structures. Change language handling. Rewrote OWL section based on input from: - University of Manchester, UK, - University Seville, Spain. Various changes to assertions due to input from the DSTC. Detailed review from Clinical Information Project, Australia. **Remove UML models to “Archetype Object Model” document.** Detailed review from UCL. **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 A Rector R Qamar I Román Martínez A Goodchild Z Z Tun E Browne T Beale T Austin T Beale K Atalag	15 Nov 2004
RELEASE 0.9			
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

Issue	Details	Raiser	Completed
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

Acknowledgements

The work reported in this document was funded by Ocean Informatics.

Thanks to Sebastian Garde, Central Qld University, Australia, for German translations.

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). ADL is designed as an abstract human-readable and computer-processible syntax. ADL archetypes can be hand-edited using a normal text editor.

The intended audience includes:

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

1.2 Related Documents

Prerequisite documents for reading this document include:

- The *open*EHR Architecture Overview

Related documents include:

- The *open*EHR Archetype Object Model (AOM) specification
- The *open*EHR Archetype Profile (oAP) specification
- The *open*EHR Template specification

1.3 Nomenclature

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

1.4 Status

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

This document is available at `http://www.openehr.org/svn/specification/Releases/1.0.2/publishing/architecture/am/adl.pdf`.

The latest version of this document can be found in PDF format at http://www.openehr.org/svn/specification/TRUNK/publishing/architecture/am/adl.pdf. New versions are announced on `openehr-announce@openehr.org`.

1.5 Peer review

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

TBD_1: (example To Be Determined paragraph)

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

To Be Continued: more work required

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

1.6 Conformance

Conformance of a data or software artefact to an *open*EHR Reference Model specification is determined by a formal test of that artefact against the relevant *open*EHR Implementation Technology Specification(s) (ITSs), such as an IDL interface or an XML-schema. Since ITSs are formal, automated derivations from the Reference Model, ITS conformance indicates RM conformance.

1.7 Changes From Previous Versions

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

1.7.1 ADL 1.5

Changes

The changes in version 1.5 are made to better facilitate the representation of specialised archetypes. The key semantic capability for specialised archetypes is to be able to support a differential representation, i.e. to express a specialised archetype only in terms of the changed or new elements in its defnition, rather than including a copy of unchanged elements. Doing the latter is clearly unsustainable in terms of change management. ADL 1.4 already supported differential representation, but somewhat inconveniently.

The changes for ADL 1.5 include:

- .adls files are introduced as the standard file extension for differential ADL files (.adl files are retained for standalone, inheritance-flattened, or 'flat', archetype);
- optional 'generated' marker in the archetype first line;
- the semantics of reference model subtype matching are now described;
- rules are provided for when node identifiers (at-codes) are required in archetypes;
- the ability to state a path rather than just an attribute, allowing redefinition blocks deep in the structure to be stated with respect to a path rather than having to be nested within numerous containing blocks on which no redefinition occurs;
- new keywords for defining the order of specialised object nodes within container attributes;
- an explanation of how to use the negated match operator (~matches, or $\not\in$) to define value set exclusions in specialised archetypes;
- rules for the construction of node identifier at-codes in specialised archetypes;

- a description of the semantics of 'inheritance-flattened' archetypes;
- optional annotations section added to archetypes;
- 'declarations' and 'invariants' sections merged into 'rules' section;
- In the ADL grammar, the `language` section is now mandatory.

Nearly all the changes occur in the section on cADL - Constraint ADL on page 43 or the new section Specialisation on page 90.

Backward Compatibility

ADL 1.5 and the corresponding changes to the AOM do not change ADL 1.4 or invalidate any existing ADL 1.4 archetypes; they just add further semantic validation rules. ADL 1.4 archetypes are therefore *syntactically* valid under ADL 1.5, but some of them may fail second pass validation, which is useful, because the ADL 1.5 validation rules detect errors that went previously undetected.

ADL 1.5 contains two pieces of new syntax: object path navigation to constraints, and ordering markers, both of which only apply in source files, not flat files. In ADL 1.5, flat (.adl) files look just the same as previously. An ADL 1.5 capable tool will generate .adl files consumable by older tools.

1.7.2 ADL 1.4

A number of small changes were made in this version, along with significant tightening up of the explanatory text and examples.

ISO 8601 Date/Time Conformance

All ISO 8601 date, time, date/time and duration values in dADL are now conformant (previously the usage of the 'T' separator was not correct). Constraint patterns in cADL for dates, times and date/times are also corrected, with a new constraint pattern for ISO 8601 durations being added. The latter allows a deviation from the standard to include the 'W' specifier, since durations with a mixture of weeks, days etc is often used in medicine.

Non-inclusive Two-sided Intervals

It is now possible to define an interval of any ordered amount (integer, real, date, time, date/time, duration) where one or both of the limits is not included, for example:

```
|0..<1000|        -- 0 >= x < 1000
|>0.5..4.0|       -- 0.5 > x <= 4.0
|>P2d..<P10d|     -- 2 days > x < 10 days
```

Occurrences for 'use_node' References

Occurrences can now be stated for use_node references, overriding the occurrences of the target node. If no occurrences is stated, the target node occurrences value is used.

Quoting Rules

The old quoting rules based on XML/ISO mnemonic patterns (&ohmgr; etc) are replaced by specifying ADL to be UTF-8 based, and any exceptions to this requiring ASCII encoding should use the "\Uhhhh" style of quoting unicode used in various programming languuages.

1.7.3 ADL 1.3

The specific changes made in version 1.3 of ADL are as follows.

Query syntax replaced by URI data type

In version 1.2 of ADL, it was possible to include an external query, using syntax of the form:

```
attr_name = <query("some_service", "some_query_string")>
```

This is now replaced by the use of URIs, which can express queries, for example:

```
attr_name = <http://some.service.org?some%20query%20etc>
```

No assumption is made about the URI; it need not be in the form of a query - it may be any kind of URI.

Top-level Invariant Section

In this version, invariants can only be defined in a top level block, in a way similar to object-oriented class definitions, rather than on every block in the definition section, as is the case in version 1.2 of ADL. This simplifies ADL and the Archetype Object Model, and makes an archetype more comprehensible as a “type” definition.

1.7.4 ADL 1.2

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 optional 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 Section

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 Sections

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 renamed to 'translations', no longer includes the original languages, and is also moved to this new top level section.

1.7.5 The Future: ADL Version 2.0

In version 2.0, the ADL syntax will be changed so that an archetype is close to being a regular dADL document. This has two consequences. Firstly, it means that special syntax such as "archetype (adl_version=1.2)" is converted to the standard object-oriented dADL tree form, and secondly, the structure of the archetype (i.e. naming of sections, design of dADL trees) is synchronised with the class definitions in the Archetype Object Model.

A full dADL form of an archetype will also be supported, in which an archetype is a faithful dADL serialisation of instances of the Archetype Object Model (AOM), allowing archetypes to be parsed as dADL documents.

Specific changes in version 2.0 will include the following.

Language section

What used to be the language section of an ADL 1.4 archetype will be adjusted to a form representing the two relevant AOM attributes, namely, `original_language` and `translations`, as per the following example:

```
    original_language = <"en">
    translations = <
        ["de"] = <
            language = <[iso_639-1::de]>
            author = <"edward.jones@translators.co.uk">
            accreditation = <"British Medical Translator id 00400595">
        >
        ["ru"] = <
```

```
                language = <[iso_639-1::ru]>
                author = <"eva.sakharova@translators.co.ru">
                accreditation = <"Russian Translator id 892230">
            >
        >
```

1.8 Tools

A validating reference ADL parser is freely available from http://www.openEHR.org. The *open*EHR ADL Workbench GUI application uses this parser and is available for all major platforms at http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/apps/doc/adl_workbench_help.htm. It has been wrapped for use in the Microsoft .Net, and standard C/C++ environments. A Java ADL parser is available at http://www.openehr.org/svn/ref_impl_java/TRUNK/docs/download.htm.

2 Overview

2.1 Context

Archetypes form the second layer of the *open*EHR semantic architecture. They provide a way of creating models of domain content, expressed in terms of constraints on a reference model. Archetypes are composed into templates, enabling them to be used for building screen forms, message schemas and other derived expressions. Archetype paths provide the basis of querying in *open*EHR as well as bindings to terminology.

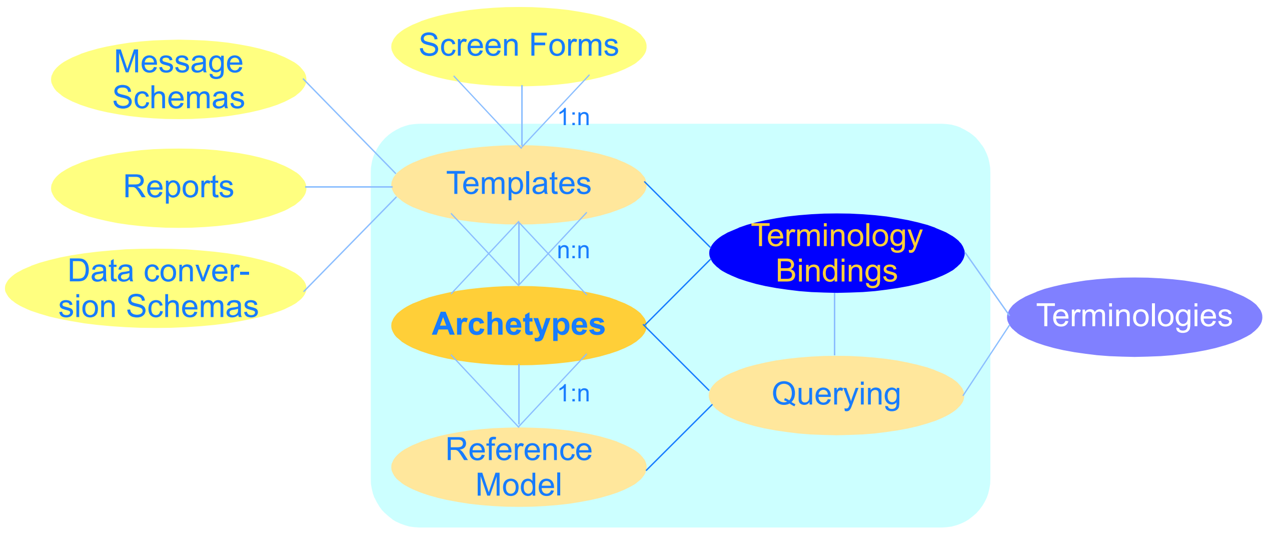


FIGURE 1 The *open*EHR Semantic Architecture

Archetypes are defined in terms of the following specifications:

- the Archetype Definition Language (ADL) - this specification;
- The openEHR Archetype Object Model (AOM);
- openEHR Archetype Profile (oAP).

The ADL syntax is semantically equivalent to the AOM as shown in FIGURE 2. ADL documents are parsed into in-memory objects (often known as a 'parse tree') which are defined by the Archetype Object Model (AOM) class definitions. The AOM can in turn be re-expressed as any number of schemas, including as a W3C XML schema. An archetype can thus be serialised as ADL or in its XML form, and parsed from either form into its object form. The XML-schema (.xsd) form is published on the *open*EHR website.

The AOM is the definitive expression of archetype semantics, and is independent of any particular syntax. The Archetype Definition Language is a formal abstract syntax for archetypes, used to provide a default serial expression of archetypes, and as the explanatory framework for most of the semantics.

The *open*EHR archetype framework is described in terms of Archetype Definitions and Principles [6] and the Archetype System [7]. Other formalisms considered in the course of the development of ADL, and some which remain relevant, are described in detailed in section Appendix A on page 110. The role of ADL with respect to XML is discussed in section 2.4 below.

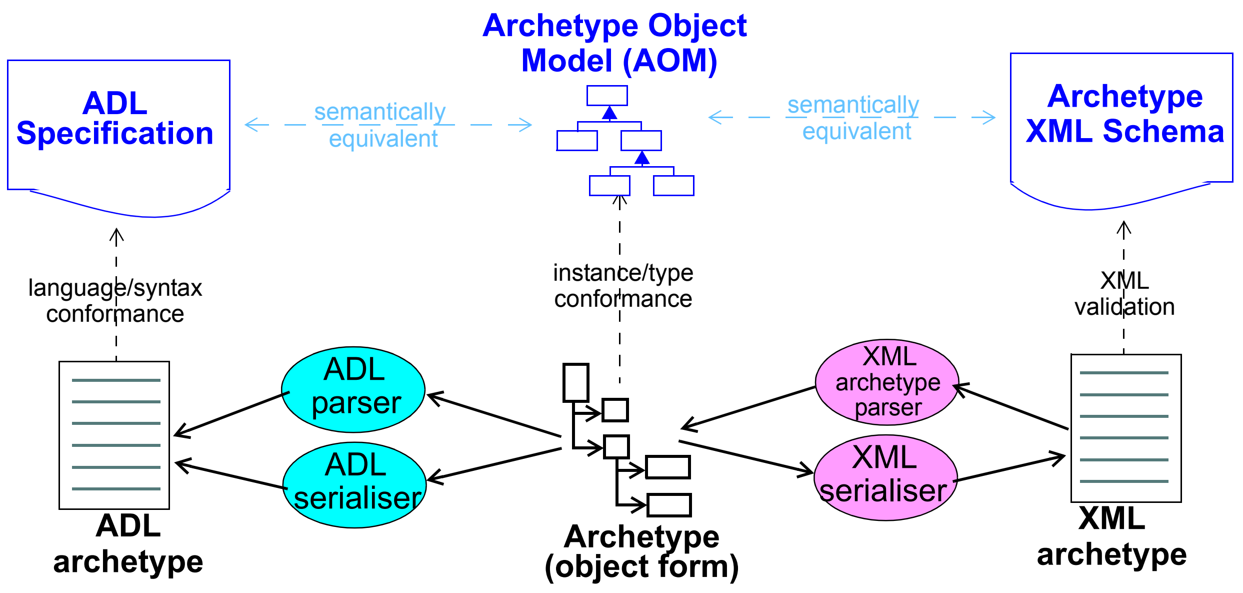


FIGURE 2 Relationship of Archetypes to Specifications

2.2 Basic Organisation

Archetypes are topic- or theme-based models of domain content, expressed in terms of constraints on a reference information model, usually denoted as the 'underlying reference model' in ths specification. Each archetype constitutes an encapsulation of a set of data points pertaining to a topic, and is therefore of a manageable, limited size, and has a clear boundary. For example a 'blood pressure measurement' archetype based on the *open*EHR reference model class `OBSERVATION` contains the data points relevant to the measurement of blood pressure.

Archetyps are 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 domain modelling, completely separate from the software. ADL can be used to write archetypes for any domain where formal object model(s) exist which describe data instances.

A 'system' of archetypes is an assembled collection of archetypes covering all or part of a domain or industry, such as clinical medicine. The constraint and object-oriented semantics of archetypes allow two kinds of relationships to be expressed between archetypes: specialisation and composition.

2.2.1 Archetype Specialisation

Any archetype can be specialised into a more refined variant via redefinition and extension. Specialised archetypes are authored in a *differential* form with respect to the parent archetype, in a similar way to how specialised classes are expressed in an object-oriented model. Differential form is a necessary pre-requisite to sustainable management of archetypes. The chain of archetypes from a specialised archetype back through all its parents the top-level ultimate parent is known as an *archetype lineage*. For non-specialised (i.e. top-level) archetypes, the lineage is just itself.

Every archetype has a 'specialisation depth'. Archetypes with no specialisation parent are depth 0, and specialised archetypes add one level to their depth for each step down a hierarchy required to reach them.

In order for specialised archetypes to be usable at runtime, the differential form used for authoring has to be compressed through the archetype lineage to create *flat-form* archetypes, i.e. the standalone equivalent of a given archetype, as if it had been constructed on its own. A flattened archetype is expressed in the same serial and object form as a differential form archetype, although there are some differences in the semantics.

Any data created via the use of an archetype conforms to the flat form of an archetype, and to the flat form of every archetype up the lineage. This notion of specialisation thus corresponds to the idea of *substitubility*, applied to data.

2.2.2 Archetype Composition

It the interests of re-use and clarity of modelling, archetypes can be composed to form larger structures semantically equivalent to a single large archetype. Composition allows two things to occur: for archetypes to be defined according to natural 'levels' or encapsulations of information, and for the re-use of smaller archetypes by higher-level archetypes. Archetype *slots* are the means of composition, and are defined in terms of constraints. Thus, unlike an object model, an archetype composition relationship is not with one fixed archetype, but instead defines a pattern for matching other suitable archetypes. Depending on what archetypes are available within the system, the archetypes matching a given slot specification can vary.

2.2.3 Computational Model

FIGURE 3 illustrates the general way archetypes are authored and validated by a 'compiler. Archetypes are created in differential, or 'source' form (left-hand side). When expressed in ADL syntax, archetype source files have the .adls extension. Where there are specialisations of an archetype, a lineage, i.e. series of related files is created. These are parsed and validated by the compiler, which converts each in an object form in memory. From these, the 'flat' form is created in memory. For a top-level archetype, this is the same as the source form, although now guaranteed valid. For each specialised archetype, the flat form is created as the result of applying its differential form over the top of the flat form of the previous specialisation level. Each flat form archetype can be serialised to a .adl file. The validator implements the rules defined in this specification. If it has access to a computable expression of the reference model underlying the archetypes, that will also be used during validation.

The above described process applies in exactly the same way for other serialised formats, such as the XML-based form.

2.3 Overview of the Formalism

2.3.1 Syntactic Structure

ADL uses three syntaxes, cADL (constraint form of ADL), dADL (data definition form of ADL), and a version of first-order predicate logic (FOPL), to express constraints on data which are instances of an underlying information model, which may be expressed in UML, relational form, or in a programming language. 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 `language`, `description`, `ontology`, and `revision_history` sections of an ADL archetype. The top-level structure of an ADL archetype is shown in FIGURE 4.

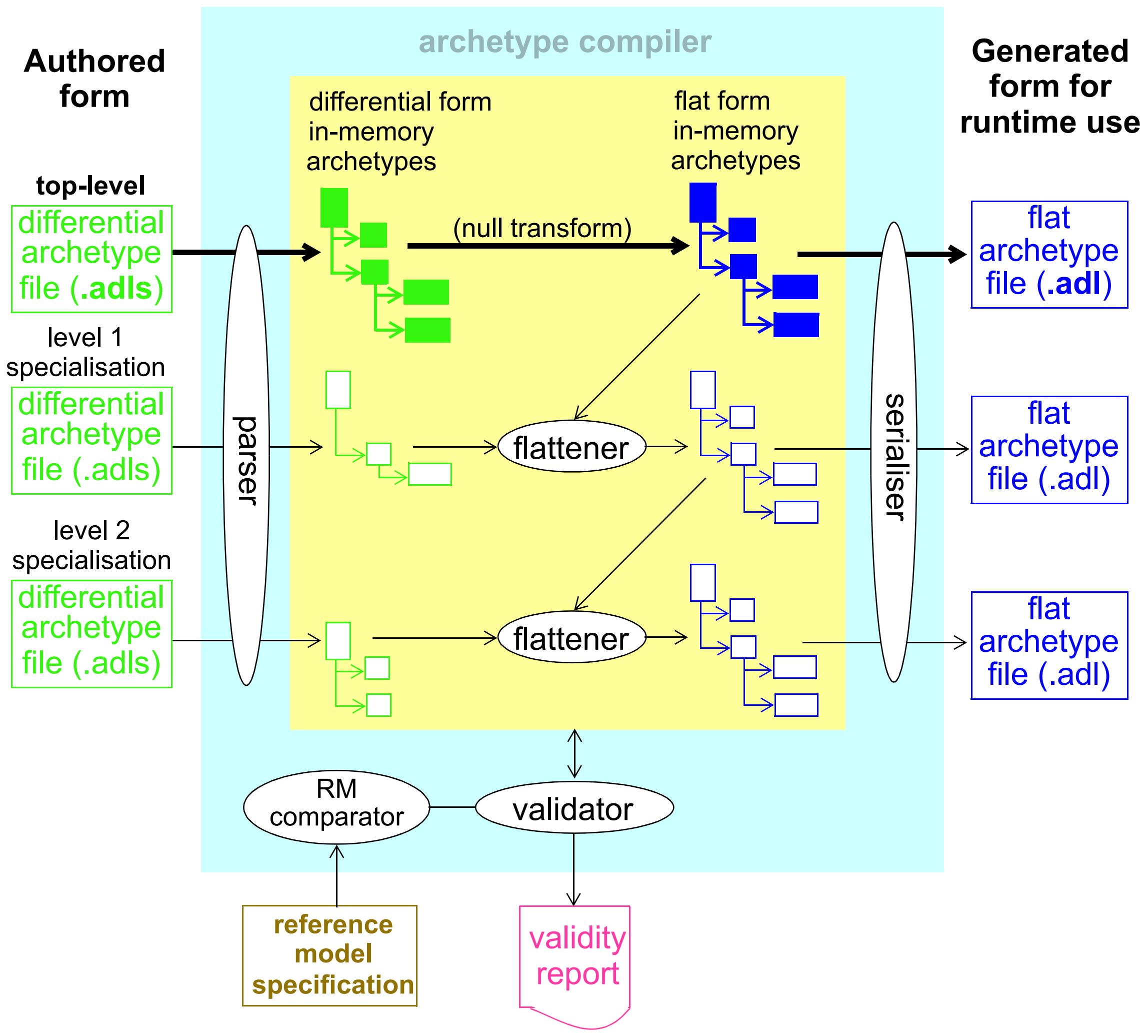

Authored form
archetype compiler
Generated form for runtime use
top-level
differential archetype file (.adls)
level 1 specialisation
differential archetype file (.adls)
level 2 specialisation
differential archetype file (.adls)
parser
differential form in-memory archetypes
flat form in-memory archetypes
(null transform)
flattener
flattener
serialiser
flat archetype file (.adl)
flat archetype file (.adl)
flat archetype file (.adl)
RM comparator
validator
reference model specification
validity report

FIGURE 3 Computational Structure of an Archetype Lineage

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

2.3.2 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=1.5)
    adl-test-instrument.guitar.draft
concept
    [at0000] -- guitar
language
    original_language = <[iso_639-1::en]>
definition
```

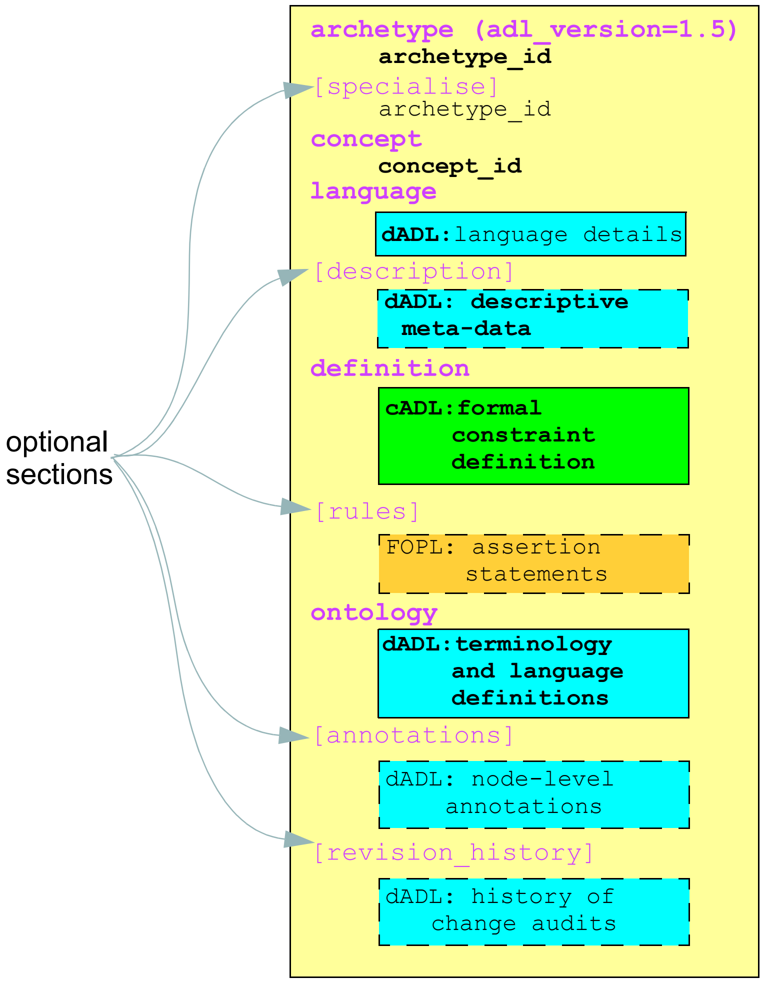

archetype (adl_version=1.5)
archetype_id
[specialise]
archetype_id
concept
concept_id
language
dADL: language details
[description]
dADL: descriptive meta-data
definition
cADL: formal constraint definition
[rules]
FOPL: assertion statements
ontology
dADL: terminology and language definitions
[annotations]
dADL: node-level annotations
[revision_history]
dADL: history of change audits
optional sections

FIGURE 4 ADL Archetype Structure

```
      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
                                      at0004]}    -- timber or nickel alloy
            }
            PART[at0002] matches {                -- body
               material matches {[local::at0003}   -- timber
            }
         }
      }
   ontology
      term_definitions = <
         ["en"] = <
            items = <
               ["at0000"] = <
                  text = <"guitar">;
                  description = <"stringed instrument">
               >
               ["at0001"] = <
                  text = <"neck">;
```

```
                        description = <"neck of guitar">
                    >
                    ["at0002"] = <
                        text = <"body">;
                        description = <"body of guitar">
                    >
                    ["at0003"] = <
                        text = <"timber">;
                        description = <"straight, seasoned timber">
                    >
                    ["at0004"] = <
                        text = <"nickel alloy">;
                        description = <"frets">
                    >
                >
            >
        >
```

2.4 The Role of ADL

While ADL is a normative syntax formalism for archetypes, the Archetype Object Model defines the semantics of an archetype, in particular relationships that must hold true between the parts of an archetype for it to be valid as a whole. Other syntaxes are therefore possible. In particular, XML-schema based XML is used for many common computing purposes relating to archetypes, and may become the dominant syntax in terms of numbers of users. Nevertheless, XML is not used as the normative syntax for archetypes for a number of reasons.

- There is no single XML schema that is likely to endure. Initially published schemas are replaced by more efficient ones, and may also be replaced by alternative XML syntaxes, e.g. the XML-schema based versions may be replaced or augmented by Schematron.
- XML has little value as an explanatory syntax for use in standards, as its own underlying syntax obscures the semantics of archetypes or any other specific purpose for which it is used. Changes in acceptable XML expressions of archetypes described above may also render examples in documents such as this obsolete.

An XML schema for archetypes is available on the *open*EHR website.

3 File Encoding and Character Quoting

3.1 File Encoding

Because ADL files are inherently likely to contain multiple languages due to internationalised authoring and translation, they must be capable of accommodating characters from any language. ADL files do not explicitly indicate an encoding because they are assumed to be in UTF-8 encoding of unicode. For ideographic and script-oriented languuages, this is a necessity.

There are three places in ADL files where non-ASCII characters can occur:

- in string values, demarcated by double quotes, e.g. “xxxx”;
- in regular expression patterns, demarcated by either // or ^^;
- in character values, demarcated by single quotes, e.g. ‘x’;

Note that URIs (a data type in dADL) are not problematic, since all characters outside the ‘unreserved set’ defined by RFC 3986[1] are already ‘percent-encoded’. The unreserved set is:

```
unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"
```

In actual fact, ADL files encoded in latin 1 (ISO-8859-1) or another variant of ISO-8859 - both containing accented characters with unicode codes outside the ASCII 0-127 range - may work perfectly well, for various reasons:

- the contain nothing but ASCII, i.e. unicode code-points 0 - 127; this will be the case in English-language authored archetypes containing no foreign words;
- some layer of the operating system is smart enough to do an on-the-fly conversion of characters above 127 into UTF-8, even if the archetype tool being used is designed for pure UTF-8 only;
- the archetype tool (or the string-processing libraries it uses) might support UTF-8 and ISO-8859 variants.

For situations where binary UTF-8 (and presumably other UTF-* encodings) cannot be supported, ASCII encoding of unicode characters above code-point 127 should only be done using the system supported by many programming languages today, namely \u escaped UTF-16. In this system, unicode codepoints are mapped to either:

- \uHHHH - 4 hex digits which will be the same (possibly 0-filled on the left) as the unicode code-point number expressed in hexadecimal; this applies to unicode codepoints in the range U+0000 - U+FFFF (the ‘base multi-lingual plane’, BMP);
- \uHHHHHHHH - 8 hex digits to encode unicode code-points in the range U+10000 through U+10FFFF (non-BMP planes); the algorithm is described in IETF RFC 2781[2].

It is not expected that the above approach will be commonly needed, and it may not be needed at all; it is preferable to find ways to ensure that native UTF-8 can be supported, since this reduces the burden for ADL parser and tool implementers. The above guidance is therefore provided only to ensure a standard approach is used for ASCII-encoded unicode, if it becomes unavoidable.

Thus, while **the only officially designated encoding for ADL and its constituent syntaxes is UTF-8**, real software systems may be more tolerant. This document therefore specifies that any tool

1. Uniform Resource Identifier (URI): Generic Syntax, Internet proposed standard, January 2005; see http://www.ietf.org/rfc/rfc3986.txt
2. see http://tools.ietf.org/html/rfc2781.

designed to process ADL files need only support UTF-8; supporting other encodings is an optional extra. This could change in the future, if required by the ADL or *open*EHR user community.

3.2 Special Character Sequences

In strings and characters, characters not in the lower ASCII (0-127) range should be UTF-8 encoded, with the exception of quoted single- and double quotes, and some non-printing characters, for which the following customary quoted forms are allowed (but not required):

- \r - carriage return
- \n - linefeed
- \t - tab
- \\ - backslash
- \” - literal double quote
- \’ - literal single quote

Any other character combination starting with a backslash is illiegal; to get the effect of a literal backslash, the \\ sequence should always be used.

Typically in a normal string, including multi-line paragraphs as used in dADL, only \\ and \” are likely to be necessary, since all of the others can be accommodated in their literal forms; the same goes for single characters - only \\ and \’ are likely to commonly occur. However, some authors may prefer to use \n and \t to make intended formatting clearer, or to allow for text editors that do not react properly to such characters. Parsers should therefore support the above list.

In regular expressions (only used in cADL string constraints), there will typically be backslash-escaped characters from the above list as well as other patterns like \s (whitspace) and \d (decimal digit), according to the PERL regular expression specification[1]. These should not be treated as anything other than literal strings, since they are processed by a regular expression parser.

1. http://www.perldoc.com/perl5.6/pod/perlre.html

4 dADL - Data ADL

4.1 Overview

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

```
person = (List<PERSON>) <
    [01234] = <
        name = <                                    -- person's name
            forenames =     <"Sherlock">
            family_name =   <"Holmes">
            salutation =    <"Mr">
        >
        address = <                                 -- person's address
            habitation_number = <"221B">
            street_name =   <"Baker St">
            city =          <"London">
            country =       <"England">
        >
    >
    [01235] = <
        -- etc
    >
>
```

In the above the attribute names `person`, `name`, `address` etc, and the type `List<PERSON>` are all assumed to come from an information model. The `[01234]` and `[01235]` tags identify container items.

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 optional; often, 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 with 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, all complex types are expressed structurally.

A common question about dADL is why it is needed, when there is already XML? To start with, this question highlights the widespread misconception about XML, namely that because it 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 that is also machine processable; it may be thought of as an *abstract syntax for object-oriented data*. 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, particularly for container types, which XML schema languages generally do not;

- not using the confusing XML notion of 'attributes' and 'elements' to represent what are essentially object properties;
- requiring half the space of the equivalent XML.

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 described in section 4.7 on page 40.

The dADL syntax as described above has a number of useful characteristics that 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 navigating 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.

4.2 Basics

4.2.1 Scope of a dADL Document

A dADL document may contain one or more objects from the same object model.

4.2.2 Keywords

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

4.2.3 Reserved Characters

In dADL, a small number of characters are reserved and have the following meanings:

'<': open an object block;

'>': close an object block;

'=': indicate attribute value = object block;

'(', ')': type name or plug-in syntax type delimiters;

'<#': open an object block expressed in a plug-in syntax;

'#>': close an object block expressed in a plug-in syntax.

Within <> delimiters, various characters are used as follows to indicate primitive values:

'"': double quote characters are used to delimit string values;

''': single quote characters are used to delimit single character values;

'|': bar characters are used to delimit intervals;

[]: brackets are used to delimit coded terms.

4.2.4 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.

4.2.5 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. A **generic type name** (including nested forms) additionally may include commas and angle brackets, but no spaces, and must be syntactically correct as per the UML. An **attribute name** is any identifier with an initial lower case letter, followed by any combination of letters, digits and underscores. Any convention that obeys this rule is allowed.

At least two well-known conventions that are ubiquitous in information models obey the above rule. One of these is the following convention:

- type names are in all uppercase, e.g. `PERSON`, except for 'built-in' types, such as primitive types (`Integer`, `String`, `Boolean`, `Real`, `Double`) and assumed container types (`List<T>`, `Set<T>`, `Hash<T, U>`), which are in mixed case, in order to provide easy differentiation of built-in types from constructed types defined in the reference model. Built-in types are the same types assumed by UML, OCL, IDL and other similar object-oriented formalisms.
- attribute names are shown in all lowercase, e.g. `home_address`.
- in both type names and attribute names, underscores are used to represent word breaks. This convention is used to maximise the readability of this document.

Other conventions may be used, such as the common programmer's mixed-case or "camel 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.

4.2.6 Semi-colons

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

```
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 in dADL.

4.3 Paths

Because dADL data is hierarchical, and all nodes are uniquely identified, a reliable path can be determined for every node in a dADL text. The syntax of paths in dADL is the standard ADL path syntax, described in detail in section 7 on page 25. Paths are directly convertible to XPath expressions for use in XML-encoded data.

A typical ADL path used to refer to a node in a dADL text is as follows.

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

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

4.4 Structure

4.4.1 General Form

A dADL document expresses serialised instances of one or more complex objects. Each such instance is a hierarchy of attribute names and object values. In its simplest form, a dADL text consists of repetitions of the following pattern:

```
attribute_name = <value>
```

In the most basic form of dADL, each attribute name is the name of an attribute in an implied or actual object or relational model. Each “value” is either a literal value of a primitive type (see Primitive Types on page 35) 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 names must be unique.

The following shows a typical structure.

```
attr_1 = <
    attr_2 = <
       attr_3 = <leaf_value>
       attr_4 = <leaf_value>
    >
    attr_5 = <
       attr_3 = <
          attr_6 = <leaf_value>
       >
       attr_7 = <leaf_value>
    >
>
attr_8 = <>
```

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). Associations between instances in dADL are also representable by references, and are described in section 4.4.6 on page 34.

Validity rules for object-structuring of dADL are as follows:

VDATU: attribute name uniqueness: sibling attributes occurring within an object node must be uniquely named with respect to each other, in the same way as for class definitions in an object reference model.

4.4.1.1 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:

```
<
    attr_1 = <
    >
    attr_8 = <>
>
```

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

4.4.1.2 Paths

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

```
/attr_1
/attr_1/attr_2
/attr_1/attr_2/attr_3        -- path to a leaf value
/attr_1/attr_2/attr_4        -- path to a leaf value
/attr_1/attr_5
/attr_1/attr_5/attr_3
/attr_1/attr_5/attr_3/attr_6 -- path to a leaf value
/attr_1/attr_5/attr_7        -- path to a leaf value
/attr_8
```

4.4.2 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.

4.4.3 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 in dADL. 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 Built-in Types on page 39 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, a different syntax is needed. Consider by way of example that an instance of the container `List<Person>` 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 inside the outer block. However, this makes the data hard to read, and does not provide an easy way of constructing paths to the contained items. A better syntax becomes more obvious when we consider that members of container objects in their computable form are nearly always accessed by a method such as `member(i)`, `item[i]` or just plain `[i]`, in the case of array access in the C-based languages. dADL opts for the array-style syntax, known in dADL as container member *keys*. No attribute name is explicitly given (see Syntax Alternatives on page 41 for further discussion of this choice); any primitive comparable value is allowed as the key, rather than just integers used in C-style array access. Further, if integers are used, it is not assumed that they dictate ordinal indexing, i.e. it is possible to use a series of keys `[2]`, `[4]`, `[8]` etc. The following example shows one version of the above container in valid dADL:

```
people = <
    [1] = <name = <> birth_date = <> interests = <>>
    [2] = <name = <> birth_date = <> interests = <>>
    [3] = <name = <> birth_date = <> interests = <>>
>
```

Strings and dates may also be used. Keys are coloured blue in the this specification in order to distinguish the run-time status of key values from the design-time status of class and attribute names. The following example shows the use of string values as keys for the contained items.

```
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 = <>>
>
```

The syntax for primitive values used as keys follows exactly the same syntax described below for data of primitive types. It is convenient in some cases to construct key values 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 guarantee 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 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 key
               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", "satire">
               weighting = <78%>
            >
         >
```

Container instances are expressed using repetitions of a block introduced by a *key*, in the form of a primitive value in brackets i.e. '[]'.

The example above conforms directly to the object-oriented type specification (given in a pascal-like syntax):

```
    class SCHEDULE
        lesson_times: List<Time>
        locations: List<String>
        subjects: List<SUBJECT> -- or it could be Hash<SUBJECT>
    end

    class SUBJECT
        name: String
        teacher: String
        topics: List<String>
        weighting: Real
    end
```

Other class specifications corresponding to the same data are possible, but will all be isomorphic to the above.

How key values relate to a particular object structure depends on the class model of objects being created due to a dADL parsing process. It is possible to write a parser which makes reasonable inferences from a class model whose instances are represented as dADL text; it is also possible to include explicit typing information in the dADL itself (see Adding Type Information below).

The validity rule for objects within a container attribute is as follows:

VDOBU: object identifier uniqueness: sibling objects occurring within a container attribute must be uniquely identified with respect to each other.

4.4.3.1 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"
```

4.4.4 Nested Container Objects

In some cases the data of interest are instances of nested container types, such as `List<List<Message>>` (a list of Message lists) or `Hash<List<Integer>, String>` (a hash of integer lists keyed by strings). The dADL syntax for such structures follows directly from the syntax for a single container object. The following example shows an instance of the type `List<List<String>>` expressed in dADL syntax.

```
list_of_string_lists = <
    [1] = <
       [1] = <"first string in first list">
       [2] = <"second string in first list">
    >
    [2] = <
       [1] = <"first string in second list">
       [2] = <"second string in second list">
       [3] = <"third string in second list">
    >
    [3] = <
       [1] = <"only string in third list">
    >
>
```

4.4.4.1 Paths

The paths of the above example are as follows:

```
/list_of_string_lists[1]/[1]
/list_of_string_lists[1]/[2]
/list_of_string_lists[2]/[1]
etc
```

4.4.5 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. the object or relational model 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 using a syntactical addition inspired by the `(type)` casting operator of the C language, whose meaning is approximately: force the type of the result of the following expression to be `type`. In dADL typing is therefore done by including the type name in parentheses 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.

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 **`org.openehr.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 in parentheses. Dot-separated namespace identifiers and template parameters may be used.

4.4.6 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 also the 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 (i.e. more general type) of the type of the actual object being referred to.

Data in other dADL documents can be referred to using the URI syntax to locate the document, with the internal path included as described above.

> **Shared objects** are referenced using paths. Objects in other dADL documents can be referred to using normal URIs whose path section conforms to dADL path syntax.

4.4.6.1 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"]
```

4.5 Leaf Data - Built-in Types

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>
```

4.5.1 Primitive Types

4.5.1.1 Character Data

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

```
'a'
```

Characters outside the low ASCII (0-127) range must be UTF-8 encoded, with a small number of backslash-quoted ASCII characters allowed, as described in File Encoding and Character Quoting on page 24.

4.5.1.2 String Data

All strings are enclosed in double quotes, as follows:

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

Quotes are encoded using ISO/IEC 10646 codes, e.g. :

```
"this is a much longer string, what one might call a &quot;phrase&quot;."
```

Line extension of strings is done simply by including returns in the string. The exact contents of the string are computed as being the characters between the double quote characters, with the removal of white space leaders up to the left-most character of the first line of the string. This has the effect of allowing the inclusion of multi-line strings in dADL texts, in their most natural human-readable form, e.g.:

```
    text = <"And now the STORM-BLAST came, and he
             Was tyrannous and strong :
             He struck with his o'ertaking wings,
             And chased us south along.">
```

String data can be used to contain almost any other kind of data, which is intended to be parsed as some other formalism. Characters outside the low ASCII (0-127) range must be UTF-8 encoded, with a small number of backslash-quoted ASCII characters allowed, as described in File Encoding and Character Quoting on page 24.

4.5.1.3 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 4.5.4 below).

4.5.1.4 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 distinguish list items (see section 4.5.4 below).

4.5.1.5 Boolean Data

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

```
True
False
```

4.5.1.6 Dates and Times

Complete Date/Times

In dADL, full and partial dates, times and durations can be expressed. All full dates, times and durations are expressed using a subset of ISO8601. The Support IM provides a full explanation of the ISO8601 semantics supported in openEHR.

In dADL, the use of ISO 8601 allows extended form only (i.e. ':' and '-' must be used). The ISO 8601 method of representing partial dates consisting of a single year number, and partial times consisting of hours only are not supported, since they are ambiguous. See below for partial forms.

Patterns for complete dates and times in dADL include the following:

```
yyyy-MM-dd                           -- a date
hh:mm:ss[,sss][Z|+/-hhmm]            -- a time with optional seconds
yyyy-MM-ddThh: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: 'Y' for years, "M' for months, 'W' for weeks, 'D' for days, 'H' for hours, 'M' for minutes, and 'S' for seconds. The literal 'T' separates the YMWD part from the HMS part, ensuring that months and minutes can be distinguished. Examples of date/time data include:

```
1919-01-23               -- birthdate of Django Reinhardt
16:35:04,5               -- rise of Venus in Sydney on 24 Jul 2003
2001-05-12T07:35:20+1000 -- timestamp on an email received from Australia
P22D4TH15M0S             -- period of 22 days, 4 hours, 15 minutes
```

Partial Date/Times

Two ways of expressing partial (i.e. incomplete) date/times are supported in dADL. The ISO 8601 incomplete formats are supported in extended form only (i.e. with '-' and ':' separators) for all patterns that are unambiguous on their own. Dates consisting of only the year, and times consisting of only the hour are not supported, since both of these syntactically look like integers. The supported ISO 8601 patterns are as follows:

```
yyyy-MM                              -- a date with no days
hh:mm                                -- a time with no seconds
yyyy-MM-ddThh:mm                     -- a date/time with no seconds
yyyy-MM-ddThh                        -- a date/time, no minutes or seconds
```

To deal with the limitations of ISO 8601 partial patterns in a context-free parsing environment, a second form of pattern is supported in dADL, based on ISO 8601. In this form, '?' characters are substituted for missing digits. Valid partial dates follow the patterns:

```
yyyy-MM-??               -- date with unknown day in month
yyyy-??-??               -- date with unknown month and day
```

Valid partial times follow the patterns:

```
hh:mm:??                 -- time with unknown seconds
hh:??:??                 -- time with unknown minutes and seconds
```

Valid date/times follow the patterns:

```
yyyy-MM-ddThh:mm:??      -- date/time with unknown seconds
yyyy-MM-ddThh:??:??      -- date/time with unknown minutes and seconds
yyyy-MM-ddT??:??:??      -- date/time with unknown time
```

```
yyyy-MM-??T??:??:??       -- date/time with unknown day and time
yyyy-??-??T??:??:??       -- date/time with unknown month, day and time
```

4.5.2 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|`	the two-sided range N >= x <= M;
`|N>..M|`	the two-sided range N > x <= M;
`|N..<M|`	the two-sided range N >= x <M;
`|N>..<M|`	the two-sided range N > x <M;
`|<N|`	the one-sided range x < N;
`|>N|`	the one-sided range x > N;
`|>=N|`	the one-sided range x >= N;
`|<=N|`	the one-sided range x <= N;
`|N +/-M|`	interval of N ± M.

The allowable values for N and M include any value in the range of the relevant type, as well as:

`infinity`	
`-infinity`	
`*`	equivalent to `infinity`

Examples of this syntax include:

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

4.5.3 Other Built-in Types

4.5.3.1 URIs

URI can be expressed as dADL data in the usual way found on the web, and follow the standard syntax from `http://www.ietf.org/rfc/rfc3986.txt`. Examples of URIs in dADL:

```
http://archetypes.are.us/home.html
ftp://get.this.file.com#section_5
http://www.mozilla.org/products/firefox/upgrade/?application=thunderbird
```

Encoding of special characters in URIs follows the IETF RFC 3986, as described under File Encoding and Character Quoting on page 24.

4.5.3.2 Coded Terms

Coded terms are ubiquitous in medical and clinical information, and are likely to become so in most other industries, as ontologically-based information systems and the 'semantic web' emerge. The logical structure of a coded term is simple: it consists of an identifier of a terminology, and an identifier of a code within that terminology. The dADL string representation is as follows:

```
[terminology_id::code]
```

Typical examples from clinical data:

```
[icd10AM::F60.1]              -- from ICD10AM
[snomed_ct::2004950]          -- from snomed-ct
[snomed_ct(3.1)::2004950]     -- from snomed-ct v 3.1
```

4.5.4 Lists of Built-in 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
```

4.6 Plug-in Syntaxes

Using the dADL syntax, any object structure can be serialised. In some cases, the requirement is to express some part of the structure in an abstract syntax, rather than in the more literal seriliased object form of dADL. dADL provides for this possibility by allowing the value of any object (i.e. what appears between any matching pair of <> delimiters) to be expressed in some other syntax, known as a "plug-in" syntax. Plug-in syntaxes are indicated in dADL in a similar way as typed objects, i.e. by the use of the syntax type in parentheses preceding the <> block. For a plug-in section, the <> delimiters are modified to <# #>, to allow for easier parser design, and easier recognition of such blocks by human readers. The general form is as follows:

```
attr_name = (syntax) <#
    ...
#>
```

The following example illustrates a cADL plug-in section in an archetype, which it itself a dADL document:

```
definition = (cadl) <#
    ENTRY[at0000] ∈ {               -- blood pressure measurement
        name ∈ {                    -- any synonym of BP
            CODED_TEXT ∈ {
                code ∈ {
                    CODE_PHRASE ∈ {[ac0001]}
                }
            }
        }
    }
#>
```

Clearly, many plug-in syntaxes might one day be used within dADL data; there is no guarantee that every dADL parser will support them. The general approach to parsing should be to use plug-in parsers, i.e. to obtain a parser for a plug-in syntax that can be built into the existing parser framework.

4.7 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 try take into account visual aesthetics of the XML. 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 Attributes

Single attribute nodes map to tagged nodes of the same name.

Container Attributes

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.

Nested Container Attributes

Nested 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, consider an object structure defined by the signature `countries:Hash<Hash<Hotel,String>,String>`. An instance of this in dADL looks as follows:

```
countries = <
    [“spain”] = <
       [“hotels”] = <...>
       [“attractions”] = <...>
    >
```

```
        ["egypt"] = <
            ["hotels"] = <...>
            ["attractions"] = <...>
        >
    >
```

can be mapped to the XML in which the synthesised element tag "_items" and the attribute "key" are used:

```
    <countries key="spain">
        <_items key="hotels">
            ...
        </_items>
        <_items key="attractions">
            ...
        </_items>
    <countries>
    </countries key="eqypt">
        <_items id="hotels">
            ...
        </_items>
        <_items key="attractions">
            ...
        </_items>
    </countries>
```

In this case, the dADL path `countries["spain"]/["hotels"]` will be transformed to the Xpath `countries[@key="spain"]/_items[@key="hotels"]` in order to navigate to the same element.

Type Names

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

4.8 Syntax Alternatives

WARNING:the syntax in this section is not part of dADL

4.8.1 Container Attributes

A reasonable alternative to the syntax described above for nested container objects would have been to use an arbitrary member attribute name, such as “items”, or perhaps “_items” (in order to indicate to a parser that the attribute name cannot be assumed to correspond to a real property in an object model), as well as the key for each container member, giving syntax like the following:

```
people = <
    _items[1] = <name = <> birth_date = <> interests = <>>
    _items[2] = <name = <> birth_date = <> interests = <>>
    _items[3] = <name = <> birth_date = <> interests = <>>
>
```

Additionally, with this alternative, it becomes more obvious how to include the values of other properties of container types, such as ordering, maximum size and so on, e.g.:

```
people = <
    _items[1] = <name = <> birth_date = <> interests = <>>
    _items[2] = <name = <> birth_date = <> interests = <>>
    _items[3] = <name = <> birth_date = <> interests = <>>
    _is_ordered = <True>
    _upper = <200>
>
```

Again, since the names of such properties in any given object technology cannot be assumed, the special underscore form of attribute names is used.

However, we are now led to somewhat clumsy paths, where “_items” will occur very frequently, due to the ubiquity of containers in real data:

```
/people/_items[1]/
/people/_items[2]/
/people/_items[3]/
/people/_is_ordered/
/people/_upper/
```

A compromise which satisfies the need for correct representation of all attributes of container types and the need for brevity and comprehensibility of paths would be to make optional the “_items”, but retain other container pseudo-attributes (likely to be much more rarely used), thus:

```
people = <
    [1] = <name = <> birth_date = <> interests = <>>
    [2] = <name = <> birth_date = <> interests = <>>
    [3] = <name = <> birth_date = <> interests = <>>
    _is_ordered = <True>
    _upper = <200>
>
```

The above form leads to the following paths:

```
/people[1]/
/people[2]/
/people[3]/
/people/_is_ordered/
/people/_upper/
```

The alternative syntax in this subsection is not currently part of dADL, but could be included in the future, if there was a need to support more precise modelling of container types in dADL. If such support were to be added, it is recommended that the names of the pseudo-attributes (“_item”, “_is_ordered” etc) be based on names of appopriate container types from a recognised standard such as OMG UML, OCL or IDL.

5 cADL - Constraint ADL

5.1 Overview

cADL is a block-structured 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 configurations 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 matches {                             -- constraint on a PERSON instance
    name matches {                           -- constraint on PERSON.name
       TEXT matches {/.+/}                   -- any non-empty string
    }
    addresses cardinality matches {1..*} matches { -- constraint on
       ADDRESS matches {                     -- PERSON.addresses
          -- etc --
       }
    }
}
```

Some of the textual keywords in this example can be more efficiently rendered using common mathematical logic symbols. In the following example, the `matches` keyword have been replaced by an equivalent symbol:

```
PERSON ∈ {                                   -- constraint on a PERSON instance
    name ∈ {                                 -- constraint on PERSON.name
       TEXT ∈ {/..*/}                        -- any non-empty string
    }
    addresses cardinality ∈ {1..*} ∈ { -- constraint on
       ADDRESS ∈ {                           -- PERSON.addresses
          -- etc --
       }
    }
}
```

The full set of equivalences appears below. Raw cADL is persisted in the text-based form, to remove any difficulties when authoring cADL text in normal text editors, and to aid reading in English. However, the symbolic form might be more widely used for display purposes and in more sophisticated tools, as it is more succinct and less language-dependent. 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. This document uses both conventions.

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 section dADL - Data ADL on page 26.

5.2 Basics

5.2.1 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]
- `include, exclude`
- `before, after`

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

Textual Rendering	Symbolic Rendering	Meaning
`matches, is_in`	∈	Set membership, "p is in P"
`not, ~`	~	Negation, "not p"
`infinitiy`	*	Infinity, 'any number of...'

Keywords are shown in blue in this document.

5.2.2 Block / Node 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 ∈ {
    name ∈ {
       PERSON_NAME ∈ {
          forenames cardinality ∈ {1..*} ∈ {/.+/}
          family_name ∈ {/.+/}
          title ∈ {"Dr", "Miss", "Mrs", "Mr", ...}
       }
    }
    addresses cardinality ∈ {1..*} ∈ {
       LOCATION_ADDRESS[at0003] ∈ {
          street_number existence ∈ {0..1} ∈ {/.+/}
          street_name ∈ {/.+/}
          locality ∈ {/.+/}
          post_code ∈ {/.+/}
          state ∈ {/.+/}
          country ∈ {/.+/}
       }
    }
}
```

In the above, an identifier (shown in green in this document) 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 match the line at the beginning of the block).

1. 'use_archetype' is deprecated

Two kinds of identifiers from the underlying information model are used, in alternation: type names (shown in upper case in this document) and attribute names (shown in lower case).

Blocks introduced by a type name are known as *object blocks* or *object nodes*, while those introduced by an attribute name are *attribute blocks* or *attribute nodes* as illustrated below.

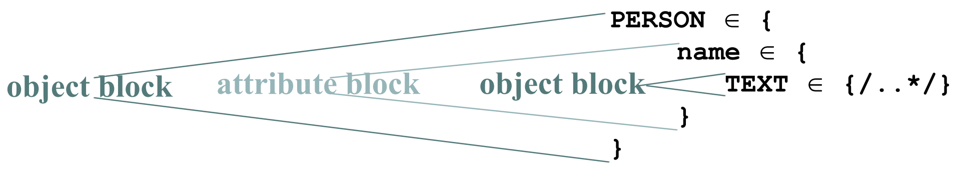


FIGURE 5 Object and Attribute Blocks in cADL

An object block or node can be thought of as a constraint matching a set of instances of the type which introduces 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 recursive nesting of constraints on types, followed by constraints on attributes (of that type), followed by types (being the types of the attribute under which it appears) until leaf nodes are reached.

A **cADL text** is a structure of alternating object and attribute blocks each introduced respectively by type names and attribute names from an underlying information model.

5.2.3 Comments

In a cADL text, comments are defined as follows:

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.

5.2.4 The Underlying Information Model

Identifiers in cADL texts correspond to entities - types and attributes - in an information model. The latter is typically an object-oriented model, but may just as easily be an entity-relationship model or any other typed model of information. A UML model compatible with the example above is shown in FIGURE 6. Note that there can be more than one model compatible with a given fragment of cADL syntax, and in particular, there are usually 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 that 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 6, 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 COM-

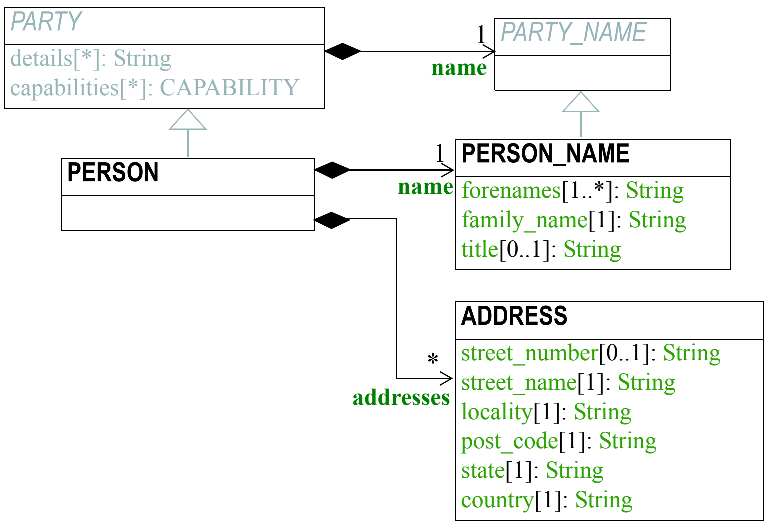


FIGURE 6 UML Model of PERSON

PANY, 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. The following example shows 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.

```
ELEMENT[at0010] matches {    -- diastolic blood pressure
    value matches {
        QUANTITY matches {
            magnitude matches {|0..1000|}
            property matches {"pressure"}
            units matches {"mm[Hg]"}
        }
    }
}
```

In this specification, the terms *underlying information model* and *reference model* are equivalent and refer to the information model on which a cADL text is based.

5.2.5 Information Model Identifiers

Identifiers from the underlying information model are used to introduce 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, names of types and the name of any property (i.e. attribute or parameterless function) can be used.

A **type name** is any identifier with an initial upper case letter, followed by any combination of letters, digits and underscores. A **generic type name** (including nested forms) additionally may include commas and angle brackets, but no spaces, and must be syntactically correct as per the OMG UML 2.x specification or higher. An **attribute name** is any identifier with an initial lower case letter, followed by any combination of letters, digits and underscores. Any convention that obeys this rule is allowed.

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 used to improve 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 based on that used in the underlying information model.

5.2.6 Node Identifiers

In cADL, an entity in brackets of the form [atNNNN] following a type name is used to identify an object node, i.e. a node constraint delimiting a set of instances of the type as defined by the reference model. Object nodes always commence with a type name. Although any node identifier format could be supported, the current version of ADL assumes that node identifiers are of the form of an archetype term identifier, i.e. [atNNNN], e.g. [at0042][1]. Node identifiers are shown in magenta in this document.

The structural function of node identifiers is to allow the formation of paths:

- enable cADL nodes in an archetype definition to be unambiguously referred to within the same archetype;
- enable data created using a given archetype to be matched at runtime;
- to enable cADL nodes in a parent archetype to be unambiguously referred to from a specialised child archetype.

Not all object nodes need node identifiers. The rules for node identifiers are given in section 5.3.11 on page 61.

> **A Node identifier** is required for any object node that is intended to be addressable elsewhere in the same archetype, in a specialised child archetype, or in the runtime data and which would otherwise be ambiguous due to sibling object nodes.

The node identifier also performs a semantic function, that of giving a design-time *meaning* to the node, by equating the node identifier to some description. The use of node identifiers in archetypes is the main source of their expressive power. Each node identifier acts as a 'semantic marker' or 'override' on the node. Thus, in the example shown in section 5.2.4, the ELEMENT node is identified by the code [at0010], which can be designated elsewhere in an archetype as meaning "diastolic blood pressure". In this way rich meaning is given to data constructed from a limited number of object types.

5.2.7 The matches Operator

The matches or is_in operator deserves special mention, since it is the key operator in cADL. This operator can be understood mathematically as set membership. When it occurs between an identifier 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 instances of a type. 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 {/[^\s\n\t]+ion[\s\n\t]/}-- the set of words ending in 'ion'
```

The following example links the name of a type XXX with a hierarchical value space.

```
XXX matches {
    aaa matches {
       YYY matches {0..3}
```

1. Note also that most *open*EHR archetype tools assume the 'atNNNN' style of identifier.

```
            }
            bbb matches {
                ZZZ matches {>1992-12-01}
            }
        }
```

The meaning of the syntax above is: data matching the constraints conssists of an instance of type `XXX`, or any subtype allowed by the underlying information model, for which the value of attribute *aaa* is of type `YYY`, or any subtype allowed by the underlying information model, and so on, recursively until leaf level constraints are reached.

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 the attribute `aaa` to any number except 5:

```
aaa ~matches {5}
aaa ~is_in {5}
aaa ∉{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`.

5.2.8 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 are generated from the archetype ontology in the language of the locale.

5.3 Constraints on Complex types

This section describes the semantics for constraining objects of complex, i.e. non-primitive types. The semantics apply recursively through a constraint structure until leaf nodes constraining primitive types are reached.

5.3.1 Attribute Constraints

In any information model, attributes are either single-valued or multiply-valued, i.e. of a generic container type such as `List<Contact>`. Both have *existence*, while multiply-valued attributes also have *cardinality*. It is the absence or presence of the cardinality constraint in cADL which indicates that the attribute being constrained is single-valued or a container attribute respectively.

5.3.1.1 Existence

The only constraint that applies to all attributes is to do with existence. An existence constraint indicates whether an attribute value is mandatory or optional, and is indicated by “0..1” or “1” markers at line ends in UML diagrams (and often mistakenly referred to as a “cardinality of 1..1”). Attributes defined in the reference model have an effective existence constraint, defined by the invariants (or lack thereof) of the relevant class. For example, the *protocol* attribute in the *open*EHR `OBSERVATION` class[1] is defined in the reference model as being optional. An archetype may redefine this to {1..1}, making the attribute mandatory. Existence constraints are expressed in cADL as follows:

```
OBSERVATION matches {
```

1. See http://www.openehr.org/releases/1.0.1/architecture/rm/ehr_im.pdf

```
        protocol existence matches {1..1} matches {
            -- details
        }
    }
```

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 type 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 can be used to indicate that an attribute must not be present in the particular situation modelled by the archetype. This may be reasonable in some cases.

5.3.2 Single-valued Attributes

A single-valued attribute is an attribute whose type as declared in the underlying class model is of a single object type rather than a container type such as a list or set. Single-valued and multiply-valued attributes are syntactically distinguished in cADL by the *cardinality* clause (see below) appearing only on the latter. Single-valued attributes can be constrained with a single object constraint as shown in the following example.

```
    items cardinality matches {*} matches
        ELEMENT[at0004] matches {                -- speed limit
            value matches {
                QUANTITY matches {
                    magnitude matches {|0..55|}
                    property matches {"velocity"}
                    units matches {"mph"}
                }
            }
        }
    }
```

Multiple *alternative* object constraints can also be defined, using a number of sibling blocks, as shown in the following example. Each block defines an alternative constraint, only one of which needs to be matched by the data. Note that node identifiers are now required to distinguish sibling blocks, according to the rules described below in section 5.3.11.

```
    items cardinality matches {*} matches
        ELEMENT[at0004] matches {                -- speed limit
            value matches {
                QUANTITY[at0022] matches {       -- miles per hour
                    magnitude matches {|0..55|}
                    property matches {"velocity"}
                    units matches {"mph"}
                }
                QUANTITY[at0023] matches {       -- km per hour
                    magnitude matches {|0..100|}
                    property matches {"velocity"}
                    units matches {"km/h"}
                }
            }
        }
    }
```

Here the occurrences of both `QUANTITY` constraints is not stated, leading to the result that only one `QUANTITY` instance can appear in runtime data, matching either one of the constraints.

> Two or more object constraints introduced by type names appearing after a single-valued attribute (i.e. one for which there is no cardinality constraint) are taken to be **alternative constraints**, only one of which is matched by the data.

5.3.3 Container Attributes

5.3.3.1 Cardinality

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

```
    HISTORY occurrences ∈ {1} ∈ {
        periodic ∈ {False}
        events cardinality ∈ {*} ∈ {
            EVENT[at0002] occurrences ∈ {0..1} ∈ {}   -- 1 min sample
            EVENT[at0003] occurrences ∈ {0..1} ∈ {}   -- 2 min sample
            EVENT[at0004] occurrences ∈ {0..1} ∈ {}   -- 3 min sample
        }
    }
```

The `cardinality` keyword 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-unique 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 data integration, 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, the number of member items it may have in the data, and optionally, whether it has "list", "set", or "bag" semantics, via the use of the keywords ordered, unordered, unique and non-unique.

The numeric part of the cardinality contraint can take the values `{0}`, `{0..0}`, `{0..n}`, `{m..n}`, `{0..*}`, or `{*}`, or a syntactic equivalent. The first two of these constraints are unlikely to be useful, but there is no reason to prevent them. There is no default cardinality, since if none is shown, the relevant attribute is assumed to be single-valued (in the interests of uniformity in archetypes, this holds even for smarter parsers that can access the reference model and determine that the attribute is in fact a container.

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 that may be empty, as in the following:

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

5.3.3.2 Occurrences

A constraint on occurrences is used only with cADL object nodes, to indicate how many times in data an instance conforming to the constraint can occur. It is usually only defined on objects that are children of a container attribute, since by definition, the occurrences of an object that is the value of a single-valued attribute can only be 0..1 or 1..1, and this is already defined by the attribute's `existence`. However, it may be used in specialised archetypes to exclude a possibility defined in a parent archetype (see Attribute Redefinition on page 96).

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[at0002] occurrences ∈ {1..1} ∈ {} -- 1 min sample
        EVENT[at0003] occurrences ∈ {0..1} ∈ {} -- 2 min sample
        EVENT[at0004] occurrences ∈ {0..1} ∈ {} -- 3 min sample
    }
```

The following example 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[at0103] ∈ {
        kind ∈ {/tribe|family|club/}
        members cardinality ∈ {*} ∈ {
           PERSON[at0104] occurrences ∈ {1} ∈ {
              title ∈ {"head"}
              -- etc --
           }
           PERSON[at0105] occurrences ∈ {0..*} ∈ {
              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 type 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 its own, thus: `{2}`, `{*}`. A range of `{0..0}` or `{0}` indicates that no occurrences of this object are allowed in this archetype.

TBD_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.

Where cardinality constraints are used (remembering that occurrences is always there by default, if not explicitly specified), cardinality and occurrences must always be compatible.

5.3.4 “Any” Constraints

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

```
PERSON[at0001] matches {
    name existence matches {1..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[at0004] 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.

5.3.5 Reference Model Type Matching

All cADL object constraints state a type from an underlying reference model. This may be an abstract type or a concrete type. The part of the data conforming to the constraint can be of any concrete type from the reference model that conforms to the type mentioned in the constraint, i.e. the same type if it is concrete, or any subtype. Correctly evaluating data/archetype conformance is up to tools to implement, and requires access to a formal description of the reference model.

One of the consequences of subtype-based type matching is that semantics are needed for when more than one reference model subtype is declared under the same attribute node in cADL. Consider the reference model inheritance structure shown below, in which the abstract PARTY class has abstract and concrete descendants including ACTOR, ROLE, and so on.

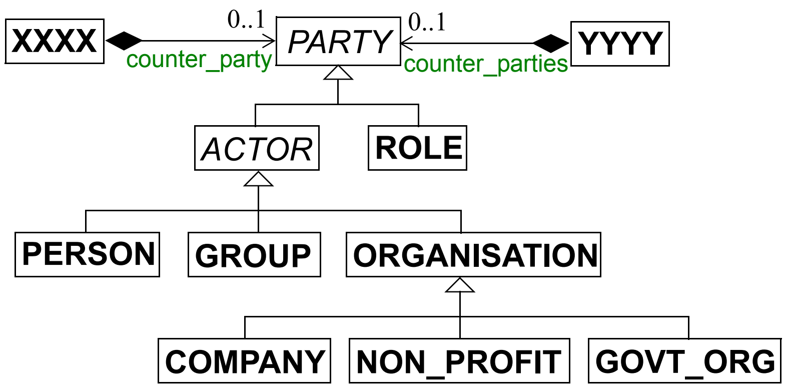


FIGURE 7 Reference model with abstract and concrete subtypes

5.3.5.1 Narrowed Subtype Constraints

The following cADL statement defines an instance space that includes any instance of any of the concrete subtypes of the PARTY class within an instance of the class XXXX in the figure.

```
    counter_party ∈ {
        PARTY ∈ { ... }
    }
```

However, in some circumstances, it may be desirable to define a constraint that will match a particular subtype in a specific way, while other subtypes are matched by the more general rule. Under a single-valued attribute, this can be done as follows:

```
    counter_party ∈ {
        PARTY ∈ { ... }
        PERSON ∈ {
            date_of_birth ∈ { ... }
        }
    }
```

This cADL text says that the instance value of the *counter_party* attribute in the data can either be a PERSON object matching the PERSON block, with a *date_of_birth* matching the given range, or else any other kind of PARTY object.

Under a multiply-valued attribute, the alternative subtypes are included as identified child members. The following example illustrates a constraint on the counter_parties attribute of instances of the class YYYY in FIGURE 7.

```
    counter_parties cardinality ∈ {*} ∈ {
        PERSON[at0004] ∈ {
            date_of_birth ∈ { ... }
        }
        ORGANISATION[at0005] ∈ {
            date_of_registration ∈ { ... }
        }
        PARTY[at0006] ∈ { ... }
    }
```

The above says that ORGANISATION and PERSON instances in the data can only match the ORGANISATION and PERSON constraints stated above, while an instance any other subtype of PARTY must match the PARTY constraint.

5.3.5.2 Remove Specified Subtypes

In some cases it is required to remove some subtypes altogether. This is achieved by stating a constraint on the specific subtypes with occurrences limited to zero. The following example matches any PARTY instance with the exception of instancs of COMPANY or GROUP subtypes.

```
    counter_party ∈ {
        PARTY ∈ { ... }
        COMPANY occurrences ∈ {0}
        GROUP occurrences ∈ {0}
    }
```

5.3.6 Paths

5.3.6.1 Archetype Path Formation

The use of object nodes allows the formation of *archetype paths*, which can be used to unambiguously reference object nodes within the same archetype or within a specialised child. The syntax of archetype paths is designed to be close to the W3C Xpath syntax, and can be directly converted to it for use in XML.

> **Archetype paths** are paths extracted from the definition section of an archetype, and refer to object nodes within the definition. A path is constructed as a concatenation of '/' characters and attribute names, with the latter including node identifiers where required for disambiguation.

In the following example, the `PERSON` constraint node does not require a node identifier, since it is the sole object constraint under the single-valued attribute *manager*:

```
manager ∈ {
   PERSON ∈ {
      title ∈ {"head of finance", "head of engineering"}
   }
}
```

The path to the object under the *title* attribute is

```
manager/title
```

Where there are more than one sibling node, node identifiers must be used to ensure distinct paths:

```
employees cardinality ∈ {*} ∈ {
   PERSON[at0104] ∈ {
      title ∈ {"head"}
   }
   PERSON[at0105] matches {
      title ∈ {"member"}
   }
}
```

The paths to the respective *title* attributes are now:

```
employees[at0104]/title
employees[at0105]/title
```

The following gives another typical example:

```
HISTORY occurrences ∈ {1} ∈ {
    periodic ∈ {False}
    events cardinality ∈ {*} ∈ {
       EVENT[at0002] occurrences ∈ {0..1} ∈ {}   -- 1 min sample
       EVENT[at0003] occurrences ∈ {0..1} ∈ {}   -- 2 min sample
       EVENT[at0004] occurrences ∈ {0..1} ∈ {}   -- 3 min sample
    }
}
```

The following paths can be constructed:

```
/                        -- the HISTORY object
/periodic                -- the HISTORY.periodic attribute
/events[at0002]          -- the 1 minute event object
/events[at0003]          -- the 2 minute event object
/events[at0004]          -- the 3 minute event object
```

The above paths can all be used to reference the relevant nodes within the archetype in which they are defined, or within any specialised child archetype.

Paths used in cADL are expressed in the ADL path syntax, described in detail in section 7 on page 25. ADL paths have the same alternating object/attribute structure implied in the general hierarchical structure of cADL, obeying the pattern `TYPE/attribute/TYPE/attribute/...` .

The examples above are *physical* paths because they refer to object nodes using node identifier codes such as "at0004". Physical paths can be converted to *logical* paths using descriptive meanings for node identifiers, if defined. Thus, the following two paths might be equivalent:

```
/events[at0004]            -- the 3 minute event object
/events[3 minute event]    -- the 3 minute event object
```

5.3.6.2 External Use of Paths

None of the paths shown above are valid outside the cADL text in which they occur, since they do not include an identifier of the enclosing document, normally an archetype. To reference a cADL node in an archetype from elsewhere (e.g. another archetype or a template), that the identifier of the document itself must be prefixed to the path, as in the following example:

```
[openehr-ehr-entry.apgar-result.v1]/events[at0002]
```

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

5.3.6.3 Runtime Paths

Paths for use with runtime data can be constructed in the same way as archetype paths, and are the same except for single-valued attributes. Since in data only a single instance can appear as the value of a single-valued attribute, there is never any ambiguity in referencing it, whereas an archetype path to or through the same attribute may require a node identifier due to he possible presence of multiple alternatives. Consider the example from above:

```
items cardinality matches {*} matches
    ELEMENT[at0004] matches {                -- speed limit
        value matches {
            QUANTITY[at0022] matches {       -- miles per hour
                magnitude matches {|0..55|}
                property matches {"velocity"}
                units matches {"mph"}
            }
            QUANTITY[at0023] matches {       -- km per hour
                magnitude matches {|0..100|}
                property matches {"velocity"}
                units matches {"km/h"}
            }
        }
    }
}
```

The following archetype paths can be constructed:

```
items[at0004]/value[at0022]
items[at0004]/value[at0023]
```

These correspond to a single runtime path:

```
items[at0004]/value
```

If the node identifiers of single-valued attribute object nodes are included in data, the archetype paths should also function correctly.

5.3.7 Internal References

It occurs reasonably often that one needs to include a constraint which is a repeat of an earlier complex constraint. This is achieved using an *archetype internal reference*, using the following syntax:

```
    use_node TYPE archetype_path
```

This statement says: use the node of type `TYPE`, found at path `archetype_path`.

> An **archetype internal reference** allows previously defined constraints to be re-used elsewhere within the same archetype or a specialised child.

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 ∈ {
    identities ∈ {
        -- etc --
    }
    contacts cardinality ∈ {0..*} ∈ {
        CONTACT [at0002] ∈ {                              -- home address
            purpose ∈ {-- etc --}
            addresses ∈ {-- etc --}
        }
        CONTACT [at0003] ∈ {                              -- postal address
            purpose ∈ {-- etc --}
            addresses ∈ {-- etc --}
        }
        CONTACT [at0004] ∈ {                              -- home contact
            purpose ∈ {-- etc --}
            addresses cardinality ∈ {0..*} ∈ {
                ADDRESS [at0005] ∈ {                      -- phone
                    type ∈ {-- etc --}
                    details ∈ {-- etc --}
                }
                ADDRESS [at0006] ∈ {                      -- fax
                    type ∈ {-- etc --}
                    details ∈ {-- etc --}
                }
                ADDRESS [at0007] ∈ {                      -- email
                    type ∈ {-- etc --}
                    details ∈ {-- etc --}
                }
            }
        }
        CONTACT [at0008] ∈ {                              -- work contact
            purpose ∈ {-- etc --}
            addresses cardinality ∈ {0..*} ∈ {
                use_node ADDRESS /contacts[at0004]/addresses[at0005] -- phone
                use_node ADDRESS /contacts[at0004]/addresses[at0006] -- fax
                use_node ADDRESS /contacts[at0004]/addresses[at0007] -- email
            }
        }
    }
```

The type mentioned in the `use_node` reference must always be the same type as, or a super-type of the referenced type. In most cases, it will be the same. In some cases, an archetype section might use a subtype of the type required by the reference model (e.g. in the above example, a type such as `POSTAL_ADDRESS`); a `use_node` reference to such a node can legally mention the parent type (`ADDRESS`, in the example). Whether this possibility has practical utility remains to be seen.

Like any other object node, a node defined using an internal reference has occurrences. If no occurrences is mentioned, the value of the occurrences is set to that of the referenced node (which if not explicitly mentioned will be the default occurrences). However, the occurrences can be overridden in the referring node, as in the following example which enables the specification for ‘phone’ to be re-used, but with a different occurrences constraint.

```
PERSON ∈ {
    contacts cardinality ∈ {0..*} ∈ {
       CONTACT [at0004] ∈ {                              -- home contact
          addresses cardinality ∈ {0..*} ∈ {
             ADDRESS [at0005] occurrences ∈ {1} ∈ {...}-- phone
          }
       }
       CONTACT [at0008] ∈ {                              -- work contact
          addresses cardinality ∈ {0..*} ∈ {
             use_node ADDRESS occurrences ∈ {0..*}
                   /contacts[at0004]/addresses[at0005]        -- phone
          }
       }
    }
```

As for any other object constraint node, `use_node` constraints must have a node identifier when occurring within a container attribute, or when otherwise ambiguous within a single-valued attribute. However, rather than having to state the identifier explicitly, it is inferred from the object identifier of the target of the reference. If an identifier is *required* at the position of the `use_node` reference due to the node identification rules, the reference target must must be given a node identifier if it doesn’t already have one. In the rare case where the identifier at the reference target is also used on a sibling node of the reference, the reference must have an explicit node identifier.

5.3.8 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’, or ‘chaining point’, i.e. a connection point whose allowable ‘fillers’ are constrained by a set of statements, written in the ADL assertion language (defined in section 5 on page 23).

An **archetype slot** defines a compositional chaining point in an archetype at which other archetypes can be inserted, if they match the constraint defined by the slot.

An archetype slot is introduced with the keyword `allow_archetype` and defined in terms of two lists of assertion statements defining which archetypes are allowed and/or which are excluded from filling that slot, introduced with the keywords `include` and `exclude`, respectively. The following example illustrates the general form of an archetype slot.

```
allow_archetype SECTION occurrences ∈ {0..*} ∈ {
   include
      -- constraints for inclusion
   exclude
      -- constraints for exclusion
}
```

Since archetype slots are typed, the reference model type of the allowed archetypes is already constrained. Otherwise, any assertion about a filler archetype can be made. The assertions do not constrain data in the way that other archetype statements do, instead they constrain archetypes. Two kinds of reference may be used in a slot assertion. The first is a reference to an object-oriented property of the filler archetype itself, where the property names are defined by the `ARCHETYPE` class in the Archetype Object Model. Examples include:

```
archetype_id
parent_archetype_id
short_concept_name
```

This kind of reference is usually used to constrain the allowable archetypes based on *archetype_id* or some other meta-data item (e.g. archetypes written in the same organisation). The second kind of reference is to absolute archetype paths in the `definition` section of the filler archetype. Both kinds of reference take the form of an Xpath-style path, with the distinction that paths referring to `ARCHETYPE` attributes *not* in the `definition` section do not start with a slash (this allows parsers to easily distinguish the two types of reference).

5.3.8.1 Slots based on Archetype Identifiers and Concepts

A basic kind of assertion is on the identifier of archetypes allowed in the slot. This is achieved with statements like the following in the include and exclude lists:

```
archetype_id/value ∈ {/openEHR-EHR-\.SECTION\..*\..*/}
```

It is possible to limit valid slot-fillers to a single archetype simply by stating a full archetype identifier with no wildcards; this has the effect that the choice of archetype in that slot is predetermined by the archetype and cannot be changed later. In general, however, the intention of archetypes is to provide highly re-usable models of real world content with local constraining left to templates, in which case a 'wide' slot definition is used (i.e. matches many possible archetypes).

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

```
SECTION [at2000] occurrences ∈ {0..1} ∈ {         -- objective
    items cardinality ∈ {0..*} ∈ {
        allow_archetype OBSERVATION[at2001] occurrences ∈ {0..1} ∈ {
            include
                short_concept_name ∈ {/.+/}
        }
        allow_archetype SECTION[at2002] occurrences ∈ {0..*} ∈ {
            include
                archetype_id/value ∈ {/openEHR-EHR-SECTION\..+\..+/}
            exclude
                archetype_id/value ∈
                        {/openEHR-EHR-SECTION\.patient_details\..+/}
        }
    }
}
```

Here, every constraint inside the block starting on an `allow_archetype` line contains constraints that must be met by archetypes in order to fill the slot. In the examples above, the constraints are in the form of regular expression syntax. In cADL, the PERL version of this is assumed.

5.3.8.2 Slots based on Other Constraints

Other constraints are possible as well, including that the allowed archetype must contain a certain keyword, or a certain path. The latter allows archetypes to be linked together on the basis of content. For example, under a "genetic relatives" heading in a Family History Organiser archetype, the following slot constraint might be used:

```
allow_archetype EVALUATION occurrences ∈ {0..*} matches {
    include
        short_concept_name ∈ {"risk_family_history"}
            ∧ ∃ /subject/relationship/defining_code →
                ~/subject/relationship/defining_code/
                    code_list.has([openehr::0]) -- "self"
}
```

This says that the slot allows archetypes on the `EVALUATION` class, which either have as their concept "risk_family_history" or, if there is a constraint on the subject relationship, then it may not include the code `[openehr::0]` (the *open*EHR term for "self") - i.e. it must be an archetype designed for family members rather than the subject of care herself.

Formal Semantics of include and exclude Lists

`To Be Continued:        rules for precedence of exclusions over inclusions.`

5.3.9 Placeholder Constraints

Not all constraints can be defined easily within an archetype. One common category of constraint that should be defined externally, and referenced from the archetype is the 'value set' for attributes where the values come from an external authoritative resource. In health, typical examples include 'terminology' resources such as WHO ICDx[1] and SNOMED[2] terminologies and drug databases. The need within the archetype in this case is to refer to a value set from the resource, rather than defining them inline. The following example shows how this is done in cADL, using the example of an external terminology resource:

```
ENTRY[at0000] ∈ {             -- blood pressure measurement
    name ∈ {                  -- any synonym of BP
        DV_CODED_TEXT ∈ {
            defining_code ∈ {
                CODE_PHRASE ∈ {[ac0001]}
            }
        }
    }
}
```

In the above, the constraint on `CODE_PHRASE` is set to an archetype constraint, or 'ac' code of the form `[acNNNN]`, which acts as an internal identifier of a value set that is defined in the external resource. In the `ontology` section of the archetype, this identifier can be bound to a URI indicating the set of values from the resource.

Placeholder constraints are an alternative to enumeration of some value sets within an archetype. Inline enumeration will work perfectly well in a technical sense, but has at least two limitations. Firstly, the intended set of values allowed for the attribute may change over time (e.g. as has happened with 'types of hepatitis' since 1980), and since the authoritative resource is elsewhere, the archetype has to be continually updated. With a large repository of archetypes, each containing inline coded term constraints, this approach is likely to be unsustainable and error-prone. Secondly, the best means of defining the value set is in general not likely to be via enumeration of the individual terms, but in the form of a semantic expression that can be evaluated against the external resource. This is because the value set is typically logically specified in terms of inclusions, exclusions, conjunctions and disjunctions of general categories.

Consider for example the value set logically defined as "any bacterial infection of the lung". The possible values would be codes from a target terminology, corresponding to numerous strains of pneumococcus, staphlycoccus and so on, but not including species that are never found in the lung. Rather than enumerate the list of codes corresponding to this value set (which is likely to be quite large), the archetype author is more likely to rely on semantic links within the external terminology to express the set; a query such as 'is-a bacteria and has-site lung' might be definable against the terminology such as SNOMED-CT or ICD10.

1. http://www.who.int/classifications/icd/en/
2. http://www.ihtsdo.org/

In a similar way, other value sets, including for quantitative values, are likely to be specified by queries or formal expressions, and evaluated by an external knowledge service. Examples include “any unit of pressure” and “normal range values for serum sodium”.

In such cases, expressing the placeholder constraint could be done by including the query or other formal expression directly within the archetype itself. However, experience shows that this is problematic in various ways. Firstly, there is little if any standardisation in such formal value set expressions or queries for use with knowledge services - two archetype authors could easily create competing syntactical expressions for the same logical constraint. A second problem is that errors might be made in the query expression itself, or the expression may be correct at the time of authoring, but need subsequent adjustment as the relevant knowledge resource grows and changes. The consequence of this is the same as for a value set enumerated inline - it is unlikely to be sustainable for large numbers of archetyes. These problems are not accidental: a query with respect to a terminological, ontological or other knowledge resource is most likely to be authored correctly by maintainers or experts of the knowledge resource, rather than archetype authors; it may well be altered over time due to improvements in the query formalism itself.

5.3.10 Mixed Structures

Four types of structure representing 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;
- *internal references*: any node introduced by the keyword `use_node`, followed by a type name; such nodes indicate re-use of a complex object constraint that 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 indicate a complex object constraint which is expressed in some other archetype;
- *placeholder constraints*: any node whose constraint is of the form `[acNNNN]`.

At any given node, any combination of these types can co-exist, as in the following example:

```
SECTION[at2000] ∈ {
    items cardinality ∈ {0..*; ordered} ∈ {
        ENTRY[at2001] ∈ {-- etc --}
        allow_archetype ENTRY[at0002] ∈ {-- etc --}
        use_node ENTRY [at0001]/some_path[at0004]/
        ENTRY[at2003] ∈ {-- etc --}
        use_node ENTRY /[at1002]/some_path[at1012]/
        use_node ENTRY /[at1005]/some_path[at1052]/
        ENTRY[at2004] ∈ {-- 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 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. Note also that the `ordered` keyword has been used to indicate that the list order is intended to be significant.

5.3.11 Summary of Object Node Identification Rules

As indicated in the section Node Identifiers on page 47, node identifiers are needed on some object nodes in order to disambiguate them from sibling nodes. This enables the construction of paths to any node in the cADL structure, and also allows specialised versions of nodes to be defined in specialised archetypes (see section 9 on page 90).

ADL takes a minimalist approach and does not require node identifiers where sibling object nodes can be otherwise distinguished. Node identifiers are mandatory in the following cases:

- all immediate child object nodes of an attribute defined as multiply-valued in the underlying information model (i.e. a container type such as a List<T>, Set<T> etc). This applies even if a particular archetype only specifies one child object of the attribute;
- all immediate child object nodes of single-valued attributes that are of the *same* reference model type, i.e. of the form of the example shown in section 5.3.2;
- with the exception of `use_node` constraints where the node identifier can be inferred from that of the target node.

In all other cases node identifiers are optional.

5.4 Constraints on Primitive Types

At the leaf nodes in a cADL text, constraints can be expressed on the following primitive types:

- Boolean;
- Character, String;
- Integer, Real;
- Date, Time, Date_time, Duration;
- lists and intervals of some of the above.

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

```
    some_attr matches {some_pattern}
```
rather than:

```
    some_attr matches {
        PRIMITIVE_TYPE matches {
            some_pattern
        }
    }
```
This is made possible because the syntax patterns of all primitive type constraints are mutually distinguishable, i.e. the type can always be inferred from the syntax alone. 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. Currently the cADL grammar **only supports the brief form** used in this specification since no practical reason has been identified for supporting the more verbose version. Theoretically however, there is nothing to prevent it being used in the future, or in some specialist application.

5.4.1 Constraints on String

Strings can be constrained in two ways: using a list of fixed strings, and using using a regular expression. All constraints on strings are case-sensitive.

5.4.1.1 List of Strings

A String-valued attribute can be constrained by a list of strings (using the dADL syntax for string lists), including the simple case of a single string. Examples are as follows:

```
species ∈ {"platypus"}
species ∈ {"platypus", "kangaroo"}
species ∈ {"platypus", "kangaroo", "wombat"}
```

The first example constraints the runtime value of the *species* attribute of some object to take the value "platypus"; the second constrains it be either "platypus" or "kangaroo", and so on. **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 constraining string attributes in a language independent way is with local [ac] codes. See Local Constraint Codes on page 28.

5.4.1.2 Regular Expression

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

The rules for including special characters within strings are described in File Encoding and Character Quoting on page 24.

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-linguistically 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.

5.4.2 Constraints on Integer

Integers can be constrained using a list of integer values, and using an integer interval.

5.4.2.1 List of Integers

Lists of integers expressed in the syntax from dADL (described in Lists of Built-in Types on page 39) can be used as a constraint, e.g.:

```
length matches {1000}          -- fixed value of 1000
magnitude matches {0, 5, 8}    -- any of 0, 5 or 8
```

The first constraint requires the attribute length to be 1000, while the second limits the value of magnitude to be 0, 5, or 8 only. A list may contain a single integer only:

```
magnitude matches {0}          -- matches 0
```

5.4.2.2 Interval of Integer

Integer intervals are expressed using the interval syntax from dADL (described in Intervals of Ordered Primitive Types on page 38). Examples of 2-sided intervals include:

```
length matches {|1000|}        -- point interval of 1000 (=fixed value)
length matches {|950..1050|}   -- allow 950 - 1050
length matches {|0..1000|}     -- allow 0 - 1000
length matches {|0..<1000|}    -- allow 0>= x <1000
length matches {|0>..<1000|}   -- allow 0> x <1000
length matches {|100+/-5|}     -- allow 100 +/- 5, i.e. 95 - 105
rate matches {|0..infinity|}   -- allow 0 - infinity, i.e. same as >= 0
```

Examples of one-sided intervals include:

```
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
```

5.4.3 Constraints on Real

Constraints on Real values follow exactly the same syntax as for Integers, in both list and interval forms. The only difference is that the real number values used in the constraints are indicated by the use of the decimal point and at least one succeeding digit, which may be 0. Typical examples are:

```
magnitude ∈ {5.5}                   -- list of one (fixed value)
magnitude ∈ {|5.5|}                 -- point interval (=fixed value)
magnitude ∈ {|5.5..6.0|}            -- interval
magnitude ∈ {5.5, 6.0, 6.5}         -- list
magnitude ∈ {|0.0..<1000.0|}        -- allow 0>= x <1000.0
magnitude ∈ {|<10.0|}               -- allow anything less than 10.0
magnitude ∈ {|>10.0|}               -- allow greater than 10.0
magnitude ∈ {|<=10.0|}              -- allow up to 10.0
magnitude ∈ {|>=10.0|}              -- allow 10.0 or more
magnitude ∈ {|80.0+/-12.0|}         -- allow 80 +/- 12
```

5.4.4 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}
```

5.4.5 Constraints on Character

Characters can be constrained in two ways: using a list of characters, and using a regular expression.

5.4.5.1 List of Characters

The following examples show how a character value may be constrained using a list of fixed character values. Each character is enclosed in single quotes.

```
color_name matches {'r'}
color_name matches {'r', 'g', 'b'}
```

5.4.5.2 Regular Expression

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

```
color_name matches {'[rgbcmyk]'}
color_name matches {'[^\s\t\n]'}
```

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]`

5.4.6 Constraints on Dates, Times and Durations

Dates, times, date/times and durations may all be constrained in three ways: using a list of values, using intervals, and using patterns. The first two ways allow values to be constrained to actual date, time etc values, while the last allows values to be constrained on the basis of which parts of the date, time etc are present or missing, regardless of value. The pattern method is described first, since patterns can also be used in lists and intervals.

> **NB:** for the date/time constraint type, parser writers should consider allowing the 'T' character to be optional on read but mandatory on save for some time. This is because previous versions of ADL did not include it, with the result that existing tools have created archetypes without the 'T' in date/time constraint patterns.

5.4.6.1 Date, Time and Date/Time

Patterns

Dates, times, and date/times (i.e. timestamps), can be constrained using patterns based on the ISO 8601 date/time syntax, which indicate which parts of the date or time must be supplied. A constraint pattern is formed from the abstract pattern `yyyy-mm-ddThh: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. A simplified grammar of the pattern is as follows (EBNF; all tokens shown are literals):

```
date_constraint:          yyyy - mm|??|XX - dd|??|XX
time_constraint:          hh : mm|??|XX : ss|??|XX
time_in_date_constraint:  T hh|??|XX : mm|??|XX : ss|??|XX
date_time_constraint:     date_constraint time_in_date_constraint
```

All expressions generated by this grammar must also satisfy the validity rules:

- where '??' appears in a field, only '??' or 'XX' can appear in fields to the right
- where 'XX' appears in a field, only 'XX' can appear in fields to the right

A fuller grammar can be defined to implement both the simplifed grammar and validity rules.

The following table shows the valid patterns that can be used, and the types implied by each pattern.

Implied Type	Pattern	Explanation
Date	`yyyy-mm-dd`	full date must be specified
Date	`yyyy-mm-??`	optional day; e.g. day in month forgotten
Date	`yyyy-??-??`	optional month, day; i.e. any date allowed; e.g. mental health questionnaires which include well known historical dates
Date	`yyyy-mm-XX`	mandatory month, no day
Date	`yyyy-??-XX`	optional month, no day
Time	`hh:mm:ss`	full time must be specified

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

Intervals

Dates, times and date/times can also be constrained using intervals. Each date, time etc in an interval may be a literal date, time etc value, or a value based on a pattern. In the latter case, the limit values are specified using the 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. Examples of such constraints:

```
|1995-??-XX|                          -- any partial date in 1995
|09:30:00|                            -- exactly 9:30 am
|< 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
|2004-05-20T00:00:00..2005-05-19T23:59:59| -- a date/time range
```

5.4.6.2 Duration Constraints

Patterns

Patterns based on ISO 8601 can be used to constraint duratoins in the same way as for Date/time types. The general form of a pattern is (EBNF; all tokens are literals):

```
P[Y|y][M|m][W|w][D|d][T[H|h][M|m][S|s]]
```

Note that allowing the 'W' designator to be used with the other designators corresponds to a deviation from the published ISO 8601 standard used in *open*EHR, namely:

- durations are supposed to take the form of PnnW or PnnYnnMnnDTnnHnnMnnS, but in *open*EHR, the W (week) designator can be used with the other designators, since it is very common to state durations of pregnancy as some combination of weeks and days.

The use of this pattern indicates which "slots" in an ISO duration string may be filled. Where multiple letters are supplied in a given pattern, the meaning is "or", i.e. any one or more of the slots may be supplied in the data. This syntax allows specifications like the following to be made:

```
Pd         -- a duration containing days only, e.g. P5d
Pm         -- a duration containing months only, e.g. P5m
PTm        -- a duration containing minutes only, e.g. PT5m
```

```
Pwd        -- a duration containing weeks and/or days only, e.g. P4w
PThm       -- a duration containing hours and/or minutes only, e.g. PT2h30m
```

List and Intervals

Durations can also be constrained by using absolute ISO 8601 duration values, or ranges of the same, e.g.:

```
PT1m                 -- 1 minute
P1dT8h               -- 1 day 8 hrs
|PT0m..PT1m30s|      -- Reasonable time offset of first apgar sample
```

Mixed Pattern and Interval

In some cases there is a need to be able to limit the allowed units as well as state a duration interval. This is common in obstetrics, where physicians want to be able to set an interval from say 0-50 weeks and limit the units to only weeks and days. This can be done as follows:

```
PWD/|P0W..P50W|      -- 0-50 weeks, expressed only using weeks and days
```

The general form is a pattern followed by a slash (‘/’) followed by an interval, as follows:

```
duration_pattern ‘/’ duration_interval
```

5.4.7 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, e.g. List<Integer>, Set<String> etc. As for complex types, this is indicated in cADL using the `cardinality` keyword, as follows:

```
some_attr cardinality ∈ {0..*} ∈ {some_constraint}
```

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. Any constraint described above for single-valued attributes, which is commensurate with the type of the attribute in question, may be used. However, as with complex objects, the meaning is now that every item in the list is constrained to be any one of the values implied by the constraint expression. For example,

```
speed_limits cardinality ∈ {0..*; ordered} ∈ {50, 60, 70, 80, 100, 130}
```

constrains each value in the list corresponding to the value of the attribute *speed_limits* (of type `List<Integer>`), to be any one of the values 50, 60, 70 etc.

5.4.8 Assumed Values

When archetypes are defined to have optional parts, an ability to define ‘assumed’ values is useful. For example, an archetype for the concept ‘blood pressure measurement’ might include an optional data point describing the patient position, with choices ‘lying’, ‘sitting’ and ‘standing’. Since the section is optional, data could be created according to the archetype which does not contain the protocol section. However, a blood pressure cannot be taken without the patient in some position, so clearly there could be an implied or ‘assumed’ value.

The archetype allows this to be explicitly stated so that all users/systems know what value to assume when optional items are not included in the data. Assumed values are currently definable on primitive types only, and are expressed after the constraint expression, by a semi-colon (‘;’) followed by a value of the same type as that implied by the preceding part of the constraint. The use of assumed values is illustrated here for a number of primitive types:

```
length matches {|0..1000|; 200}            -- allow 0 - 1000, assume 200
some_flag matches {True, False; True}      -- allow T or F, assume T
some_date matches {yyyy-mm-dd hh:mm:XX; 1800-01-01T00:00:00}
```

If no assumed value is stated, no reliable assumption can be made by the receiver of the archetyped data about what the values of removed optional parts might be, from inspecting the archetype. However, this usually corresponds to a situation where the assumed value does not even need to be stated - the same value will be assumed by all users of this data, if its value is not transmitted. In other cases, it may be that it doesn't matter what the assumed value is. For example, an archetype used to capture physical measurements might include a "protocol" section, which in turn can be used to record the "instrument" used to make a given measurement. In a blood pressure specialisation of this archetype it is fairly likely that physicians recording or receiving the data will not care about what instrument was used.

6 Assertions

6.1 Overview

This section describes the assertion sub-language of archetypes. Assertions are used in archetype "slot" clauses in the cADL `definition` section, and in the `rules` section. The following simple assertion in the `rules` section of an archetype says that the speed in kilometres of some node is related to the speed-in-miles by a factor of 1.6:

```
validity: /speed[at0002]/kilometres/magnitude =
          /speed[at0004]/miles/magnitude * 1.6
```

6.1.1 Requirements

Assertions are needed in archetypes to express rules in two locations in an archetype. In an archetype slot, assertions can be stated to control what archetypes are allowed in the slot, as shown in the following example:

```
CLUSTER[at0003] occurrences matches {0..1} matches {-- Detail
    items cardinality matches {0..*; unordered} matches {
        allow_archetype CLUSTER[at0009] occurrences matches {0..1} matches {
            include
                archetype_id/value matches {/openEHR-EHR-CLUSTER.exam-.+\.v1/}
        }
    }
}
```

In the above, the statement following the `include` keyword expresses a condition on the value found at the path `archetype_id/value`, using the familiar ADL `matches` operator, and a regular expression on archetype identifiers. Most slot statements are of this kind, with some requiring slightly more complex expressions. See section 5.3.8 on page 57 for more details.

The main requirement for assertions in archetypes is for expressing rules that cannot be expressed uding the standard cADL syntax. Types of rules include:

- constraints involving more than one node in an archetype, such as a rule stating that the sum of the five 0-2 value scores in an Apgar test (heartrate, breathing, muscle tone, reflex, colour) correspond to the Apgar total, recorded in a sixth node;
- rules involving predefined variables such as 'current date';
- rules involving query results from a data or knowledge context, allowing values such as 'patient date of birth' to be referenced.

The semantic requirements are for expressions including arithmetic, boolean, and relational operators, some functions, quantifier operators, a notion of operator precedence, parentheses, constant values, and certain kinds of variables. However, there is no requirement for procedural semantics, type declarations or many of the other complexities of full-blown programming languages.

6.1.2 Design Basis

The archetype assertion language is a small language of its own. Formally it is a reduced first-order predicate logic language with various operators. It has similarities with OMG's OCL (Object Constraint Language) syntax, and is also similar to the assertion syntax which has been used in the Object-Z [14] and Eiffel [12] languages and tools for over a decade (see Sowa [15], Hein [8], Kilov &

Ross [9] for an explanation of predicate logic in information modelling). None of these languages has been used directly, for reasons including:

- OCL has a complex type system, and includes some undecidable procedural semantics;
- none have adequate variable referencing mechanisms, such as to paths and external queries;
- they are too powerful, and would introduce unnecessary complexity into archetypes and templates.

There are also similarities with other languages developed in the health arena for expressing 'medical logic' (Arden), guidelines (GLIF and many others) and decision support (GELLO and many others). These languages were not directly used either, for reasons including:

- none have a path referencing mechanism;
- some are too procedural (Arden, GLIF);
- current versions of some of these languages have been made specific to the HL7v3 RIM, a particular model of health information designed for message representation (GLIF 3.x, GELLO);
- all in their published form are too powerful for the needs identified here.

The design approach used here was to create a small concrete syntax allowing for a core subset of first-order predicate logic, which could easily be parsed into a typical parse-tree form, defined in the *open*EHR Archetype Object Model. Many different variations on syntax elements are possible (as evidenced by the many formal logic syntaxes used in mathematics and computing theory); the elements used here were chosen for ease of expression using normal kebyoard characters and intuitiveness.

6.2 Keywords

The syntax of the invariant section is a subset of first-order predicate logic. In it, the following 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	$\exists$	Existential quantifier, "there exists ..."
for_all	$\forall$	Universal quantifier, "for all x..."
implies	$\rightarrow$	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, ~	~, $\neg$	Negation, "not p"

6.3 Typing

The assertion language is fully typed. All operators, variables and constants have either assumed or declared type signatures.

6.4 Operators

Assertion expressions can include arithmetic, relational and boolean operators, plus the existential and universal quantifiers.

6.4.1 Arithmetic Operators

The supported arithmetic operators are as follows:

addition: +

subtraction: -

multiplication: *

division: /

exponent: ^

modulo division: % -- remainder after integer division

6.4.2 Equality Operators

The supported equality operators are as follows:

equality: =

inequality: !=

The semantics of these operators are of value comparison.

6.4.3 Relational Operators

The supported relational operators are as follows:

less than: <

less than or equal: <=

greater than: >

greater than or equal: >=

The semantics of these operators are of value comparison on entities of Comparable types (see *open*EHR Support IM, Assumed Types section). All generate a Boolean result.

6.4.4 Boolean Operators

The supported boolean operators are as follows:

not: **not**

and: **and**

xor: **xor**

implies: **implies**

set membership: **matches**, **is_in**

The boolean operators also have symbolic equivalents shown earlier. All boolean operators take Boolean operands and generate a Boolean result. The `not` operator can be applied as a prefix operator to all operators returning a boolean result.

6.4.5 Quantifiers

The two standard logical quantifier operators are supported:

existential quantifier: **exists**

universal quantifier: **for_all**

These operators also have the usual symbolic equivalents shown earlier. The `exists` operator can be used on an variable, including paths referring to a node or value within an archetype. The `for_all` operator can be applied to sets and lists, such as referred to by a path to a multiply-valued attribute.

6.4.6 Functions

The following functions are supported:

sum(x, y, ...): equivalent to x + y + ...

mean(x, y, ...): the mean (average) value of x, y, ...

max(x, y, ...): the maximum value among x, y, ...

min(x, y, ...): the minimum value among x, y, ...

All of the above functions have the signature `func(Real, ...):Real`, but will also perform as though having the signature `func(Integer, ...):Integer`, due to automatic numeric type promotion/demotion rules.

Other functions may be added in the future.

6.5 Operands

Operands in an assertion expression are typed and are of four kinds, as described in the following sub-sections.

6.5.1 Constants

Constant values are of any primitive type defined in the *open*EHR Support IM Assumed Types, and expressed according in the dADL syntax (see section 4.5 on page 35), i.e.:

- Character, e.g. `'x'`;
- String, e.g. `"this is a string"`;
- Boolean, e.g. `True`, `False`;
- Integer, e.g. `5`;
- Real, e.g. `5.2`;
- ISO8601_DATE, e.g. `2004-08-12`;
- ISO8601_TIME, e.g. `12:00:59`;
- ISO8601_DATE_TIME, e.g. `2004-08-12T12:00:59`;
- ISO8601_DURATION, e.g. `P39W`;
- URI, e.g. `http://en.wikipedia.org/wiki/Everest`;
- coded term, e.g. `[snomed_ct::2004950]`;
- Intervals of any numeric type, according to dADL syntax e.g. `|70..130|`;

- List of any primitive type, e.g. `"string1", "string2", "string3"`;

6.5.2 Object References

A reference to an object in data, including a leaf value, is expressed using an archetype path. All such paths are absolute (i.e. contain a leading '/') and are understood to be with respect to the root of the current archetype. References to archetype nodes have the type defined at the relevant point in the underlying reference model. Examples include:

```
    /data/items[at0003]/value/value -- Date of initial onset; type ISO8601_DATE
```

To Be Continued:

6.5.3 Built-in Variables

A small number of built-in variables are available for use in assertions, and are referred to using a '$' symbol, for example `$current_date`. Built-in variables defined include:

```
    $current_date: ISO8601_DATE
    $current_time: ISO8601_TIME
    $current_date_time: ISO8601_DATE_TIME
    $current_year: Integer
    $current_month: Integer
```

6.5.4 Archetype-defined Variables

Variables may be declared within the rules section of an archetype. This is done using the following syntax:

```
    $var_name:Type ::= expression
```

This facility can be used to equate a variable name to a path, e.g. the following equates the variable `$diagnosis` to the code at the path contianing the diagnosis (e.g. in the `openEHR-EHR-EVALUATION.problem-diagnosis.v1` archetype):

```
    $diagnosis:CODE_PHRASE ::= /data/items[at0002.1]/value/defining_code
```

The variable can then be used instead of the path in subsequent expressions.

6.5.5 External Queries

An expression referring to an externally defined query, possibly including arguments, may be defined using the variable declaration syntax. The general pattern is as follows:

```
    $varname:Type ::= query(context, query_name, arg1, arg2, ...)
```

Examples include:

```
    $date_of_birth:ISO8601_DATE ::=   query("patient", "date_of_birth")
    $has_diabetes:Boolean ::=         query("patient", "has_diagnosis",
                                         "snomed_ct::1234567")
    $is_female:Boolean ::=            query("patient", "is_female")
```

Any number of arguments can be included.

Query Contexts

Query Names

6.6 Precedence and Parentheses

To Be Continued:

6.7 Conditions

Example....

```
    $is_female implies exists /path/to/xxx
```

6.8 Natural Language Issues

xx

7 ADL Paths

7.1 Overview

The notion of paths is integral to ADL, and a common path syntax is used to reference nodes in both dADL and cADL sections of an archetype. The same path syntax works for both, because both dADL and cADL have an alternating object/attribute structure. However, the interpretation of path expressions in dADL and cADL differs slightly; the differences are explained in the dADL and cADL sections of this document. This section describes only the common syntax and semantics.

The general form of the path syntax is as follows (see syntax section below for full specification):

```
path: ['/'] path_segment { '/' path_segment }+
path_segment: attr_name [ '[' object_id ']' ]
```

Essentially, ADL paths consist of segments separated by slashes ('/'), where each segment is an attribute name with optional object identifier predicate, indicated by brackets ('[]').

ADL Paths are formed from an alternation of segments made up of an attribute name and optional object node identifier predicate, separated by slash ('/') characters. Node identifiers are delimited by brackets (i.e. []).

Similarly to paths used in file systems, ADL paths are either absolute or relative, with the former being indicated by a leading slash.

Paths are **absolute** or **relative** with respect to the document in which they are mentioned. Absolute paths commence with an initial slash ('/') character.

The ADL path syntax also supports the concept of "movable" path patterns, i.e. paths that can be used to find a section anywhere in a hierarchy that matches the path pattern. Path patterns are indicated with a leading double slash ("//") as in Xpath.

Path **patterns** are absolute or relative with respect to the document in which they are mentioned. Absolute paths commence with an initial slash ('/') character.

7.2 Relationship with W3C Xpath

The ADL path syntax is semantically a subset of the Xpath query language, with a few syntactic shortcuts 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 key `child::` does not need to be used.

ADL does not distinguish attributes from children, and also assumes the `node_id` attribute. Thus, the following expressions are legal for cADL structures:

```
items[1]         -- the first member of 'items'
items[systolic]  -- the member of 'items' with meaning 'systolic'
items[at0001]    -- the member of 'items' with node id 'at0001'
```

The Xpath equivalents are:

```
items[1]         -- the first member of 'items'
items[meaning() = 'systolic']-- the member of 'items' for which the meaning()
function evaluates to "systolic"
```

```
    items[@archetype_node_id = 'at0001']-- the member of 'items' with key
    'at0001'
```

In the above, `meaning()` is a notional function is defined for Xpath in *open*EHR, which returns the rubric for the `node_id` of the current node. Such paths are only for display purposes, and paths used for computing always use the 'at' codes, e.g. `items[at0001]`, for which the Xpath equivalent is `items[@node_id = 'at0001']`.

The ADL movable path pattern is a direct analogue of the Xpath syntax abbreviation for the 'descendant' axis.

8 ADL - Archetype Definition Language

8.1 Introduction

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 Customising ADL on page 107, and the succeeding sections.

An ADL archetype follows the structure shown below:

```
archetype (...)
    archetype_id
[specialize
    parent_archetype_id]
concept
    coded_concept_name
language
    dADL language description section
description
    dADL meta-data section
definition
    cADL structural section
[rules
    assertions]
ontology
    dADL definitions section
[annotations
    dADL section]
[revision_history
    dADL section]
```

8.1.1 Global Archetype Validity

The following validity constraints apply to an archetype as a whole. Note that the term "section" means the same as "attribute" in the following, i.e. a section called "definition" in a dADL text is a serialisation of the value for the attribute of the same name.

VARID: archetype identifier validity. The archetype must have an identifier value for the archetype_id section. The identifier must conform to the published *open*EHR specification for archetype identifiers.

VARCN: archetype concept validity. The archetype must have an archetype term value in the concept section. The term must exist in the archetype ontology.

VARDF: archetype definition validity. The archetype must have a definition section, expressed as a cADL syntax string, or in an equivalent plug-in syntax.

VARON: archetype ontology validity. The archetype must have an ontology section, expressed as a dADL syntax string, or in an equivalent plug-in syntax.

8.2 File Convention

Up until ADL 1.4, archetypes expressed in ADL have been saved in .adl files. These are now known as 'flat' format files, i.e. standalone archetypes flattened through inheritance. Beginning with ADL

1.5, differential file format is supported. This is the same for top-level archetypes, but different for specialised archetypes, in that it follows the object-oriented convention and only includes overridden or new elements but does not include inherited elements. Instead, inherited elements are determined by evaluating (or 'compiling') a differential archetype with respect to its parent archetypes in an *inheritance lineage*. Differential 'source' files i.e. the files that archetype editing tools use are indicated by the extension .adls, including for non-specialised archetypes.

8.3 Basics

8.3.1 Keywords

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

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

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

Deprecated keywords include:

- `invariant -- equivalent to 'rules'`

8.3.2 Node Identifier Codes

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 by convention have 4 digits, e.g. `[at0010]`. Codes of any length are acceptable in ADL archetypes. 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.

8.3.3 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]`. As for "at" codes, the convention used in this document is to use 4-digit "ac" codes, even though any number of digits is acceptable. The use of these codes is described in section 8.7.4

8.4 Header Sections

8.4.1 Archetype Identification Section

This section introduces the archetype with the 'archetype' keyword, followed by a small number of items of meta-data in parentheses, and an archetype identifier. A typical `archetype` section is as follows:

```
archetype (adl_version=1.4)
```

```
        openEHR-EHR-ENTRY.haematology.v1
```

The multi-axial archetype identifier identifies archetypes in a global space. The syntax of the identifier is described in the Identification section of the *open*EHR Support IM specification..

8.4.1.1 Specialisation Level of the Archetype

The specialisation level of an archetype is defined by the specialisation level of its archetype identifier. A top-level identifier is level 0, with each level of specialisation increasing the specialisation level by one. The following is the header section from a specialised archetype:

```
    archetype (adl_version=1.4)
        openEHR-EHR-ENTRY.laboratory-lipds.v1
```

8.4.1.2 ADL Version Indicator

An ADL version identifier is mandatory in all archetypes, and is expressed as a string of the form adl_version=N.M, where N.M is the ADL version identifier.

8.4.1.3 Controlled Indicator

A flag indicating whether the archetype is change-controlled or not can be included after the version, as follows:

```
    archetype (adl_version=1.4; controlled)
        openEHR-EHR-ENTRY.haematology.v1
```

This flag may have the two values “controlled” and “uncontrolled” only, and is an aid to software. Archetypes that include the “controlled” flag should have the revision history section included, while those with the “uncontrolled” flag, or no flag at all, may omit the revision history. This enables archetypes to be privately edited in an early development phase without generating large revision histories of little or no value.

8.4.1.4 Generated Indicator

A flag indicating whether the archetype was generated or authored can be included after the version, as follows:

```
    archetype (adl_version=1.4; generated)
        openEHR-EHR-ENTRY.haematology.v1
```

This marker is used to support the migration to differential archetype representation introduced in ADL 1.5, to enable proper representation of specialised archetypes. The ‘generated’ marker can be used on specialised archetypes - i.e. ADL 1.5 style .adls files - generated from flat archetypes - ADL 1.4 .adl files - and also in flat archetypes generated from differential files, by an inheritance-flattening process.

8.4.2 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, i.e. an archetype cannot ‘multiply-inherit’ from other archetypes. An example of declaring specialisation is as follows:

```
    archetype (adl_version=1.4)
        openEHR-EHR-ENTRY.haematology-cbc.v1
    specialise
        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 the `ARCHETYPE_ID` definition in the identification package in the *open*EHR Support IM specification.

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

8.4.3 Concept Section

All archetypes represent some real world concept, such as a “patient”, a “blood pressure”, or an “ante-natal 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
    [at0000]    -- haematology result
```

In this concept definition, the term definition of `[at0000]` is the proper description corresponding to the “`haematology-cbc`” section of the archetype identifier above.

8.4.4 Language Section and Language Translation

The `language` section includes meta-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 `translations` list must be updated every time a translation of the archetype is undertaken. The following shows a typical example.

```
language
    original_language = <[iso_639-1::en]>
    translations = <
        [“de”] = <
            language = <[iso_639-1::de]>
            author = <
                [“name”] = <"Frederik Tyler">
                [“email”] = <"freddy@something.somewhere.co.uk">
            >
            accreditation = <"British Medical Translator id 00400595”>
        >
        [“ru”] = <
            language = <[iso_639-1::ru]>
            author = <
                [“name”] = <"Nina Alexandrovna">
                [“organisation”] = <"Dostoevsky Media Services">
                [“email”] = <"nina@translation.dms.ru">
            >
            accreditation = <"Russian Translator id 892230-3A”>
        >
    >
```

Archetypes must always be translated completely, or not at all, to be valid. This means that when a new translation is made, every language dependent section of the `description` and `ontology` sections has to be translated into the new language, and an appropriate addition made to the `translations` list in the language section.

Note: some non-conforming ADL tools in the past created archetypes without a language section, relying on the ontology section to provide the original_language (there called primary_language) and list of languages (languages_available). In the interests of backward compatibility, tool builders should consider accepting archetypes of the old form and upgrading them when parsing to the correct form, which should then be used for serialising/saving.

8.4.5 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 = <
        [“name”] = <"Dr J Joyce">
        [“organisation”] = <"NT Health Service">
        [“date”] = <2003-08-03>
    >
    lifecycle_state = <"initial">
    resource_package_uri =
        <"www.aihw.org.au/data_sets/diabetic_archetypes.html">

    details = <
        [“en”] = <
            language = <[iso_639-1::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">
            original_resource_uri =
                <"www.healthdata.org.au/data_sets/
                               diabetic_review_data_set_1.html">
            other_details = <...>
        >
        [“de”] = <
            language = <[iso_639-1::de]>
            purpose = <"Archetyp für die Untersuchung von Patienten
                     mit Diabetes">
            use = <"wird benutzt für alle Diabetes-Untersuchungen im
                  Krankenhaus, inklusive der ersten Vorstellung. Optionale
                  Abschnitte werden in Abhängigkeit von der speziellen
                  Vorstellung entfernt."
            >
            misuse = <"nicht geeignet für Benutzung vor Diagnosestellung">
            original_resource_uri =
                <"www.healthdata.org.au/data_sets/
                               diabetic_review_data_set_1.html">
            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 *open*EHR Archetype Object Model (AOM).

The `description` section is technically optional according to the AOM, but in any realistic use of ADL for archetypes, it will be required. A minimal description section satisfying to the AOM is as follows:

```
description
```

```
        original_author = <
            ["name"] = <"Dr J Joyce">
            ["organisation"] = <"NT Health Service">
            ["date"] = <2003-08-03>
        >
        lifecycle_state = <"initial">
        details = <
            ["en"] = <
                language = <[iso_639-1::en]>
                purpose = <"archetype for diabetic patient review">
            >
        >
```

8.5 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
                DV_CODED_TEXT ∈ {
                    defining_code ∈ {
                        CODE_PHRASE ∈ {[ac0001]}
                    }
                }
            }
            data ∈ {
                HISTORY[at9001] ∈ {                              -- history
                    events cardinality ∈ {1..*} ∈ {
                        EVENT[at9002] occurrences ∈ {0..1} ∈ {-- baseline
                            name ∈ {
                                DV_CODED_TEXT ∈ {
                                    defining_code ∈ {
                                        CODE_PHRASE ∈ {[ac0002]}
                                    }
                                }
                            }
                            data ∈ {
                                ITEM_LIST[at1000] ∈ {   -- systemic arterial BP
                                    items cardinality ∈ {2..*} ∈ {
                                        ELEMENT[at1100] ∈ {              -- systolic BP
                                            name ∈ {        -- any synonym of 'systolic'
                                                DV_CODED_TEXT ∈ {
                                                    defining_code ∈ {
                                                        CODE_PHRASE ∈ {[ac0002]}
                                                    }
                                                }
                                            }
                                            value ∈ {
                                                DV_QUANTITY ∈ {
                                                    magnitude ∈ {0..1000}
                                                    property ∈ {[properties::0944]}
                                                                            -- "pressure"
                                                    units ∈ {[units::387]}  -- "mm[Hg]"
                                                }
                                            }
                    }
                    ELEMENT[at1200] ∈ {          -- diastolic BP
                       name ∈ {         -- any synonym of 'diastolic'
                          DV_CODED_TEXT ∈ {
                             defining_code ∈ {
                                CODE_PHRASE ∈ {[ac0003]}
                             }
                          }
                       }
                       value ∈ {
                          DV_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`, `ITEM_LIST`, `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).

8.5.1 Design-time and Run-time paths

All non-unique sibling nodes in `definition` section that might be referred to from elsewhere in the archetype (via `use_node`), in a specialised child archetype, or might be queried at runtime, require a node identifier according to the rules described under Summary of Object Node Identification Rules on page 61. When data are created according to the `definition` section of an archetype, the archetype node identifiers can be 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 doing 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 shows the difference between design-time meanings associated with node identifiers and runtime names is the following, from a `SECTION` archetype 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 {
      DV_CODED_TEXT matches {
         defining_code matches {[ac0001]}-- any clinical problem type
      }
   }
```

In the above, the node identifier `[at0000]` is assigned a meaning such as "clinical problem" in the archetype ontology section. The subsequent 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 identifier 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 with meaning “problem”, or all nodes describing the problem “diabetes”. Internal `[acNNNN]` codes are described in Local Constraint Codes on page 78.

8.6 Rules Section

The `rules` section in an ADL archetype introduces assertions which relate to the entire archetype, and can be used to make statements which are not possible within the block structure of the `definition` section. Any constraint which relates more than one property to another is in this category, as are most constraints containing mathematical or logical formulae. Rules are expressed in the archetype assertion language, described in section 6 on page 69.

An assertion 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.

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:

```
rules
    validity: /speed[at0002]/kilometres/magnitude =
              /speed[at0004]/miles/magnitude * 1.6
```

8.7 Ontology Section

8.7.1 Overview

The `ontology` section of an archetype is expressed in dADL, and is where codes representing node IDs, constraints on text or terms, and bindings to terminologies are defined. Linguistic language translations are added in the form of extra blocks keyed by the relevant language. The following example shows the layout of this section.

```
ontology
    terminologies_available = <“snomed_ct”, ...>

    term_definitions = <
        [“en”] = <
            items = <...>
        >
        [“de”] = <
            items = <...>
        >
    >

    term_bindings = <
        [“snomed_ct”] = <
            items = <...>
        >
        ...
    >

    constraint_definitions = <
        [“en”] = <...>
```

```
            ["de"] = <
               items = <...>
            >
            ...
         >

         constraint_bindings = <
            ["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.

8.7.2 Ontology Header Statements

The terminologies_available statement includes the identifiers of all terminologies for which `term_bindings` sections have been written.

Note: some ADL tools in the past created archetypes with primary_language and languages_available statements rather than the original_languages and translations blocks in the `language` section. In the interests of backward compatibility, tool builders should consider accepting archetypes of the old form and upgrading them when parsing to the correct form, which should then be used for serialising/saving.

8.7.3 Term_definitions 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 a structure of name/value pairs, and mustat least include the names "text" and "description", which are akin to the usual rubric, and full definition found in terminologies like SNOMED-CT. Each term object is then included in the appropriate language list of term definitions, as shown in the example below.

```
   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"] = <
         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:

```
["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, only its origin. Bindings are described in section 8.7.5 and section 8.7.6.

Note: the use of "items" in the above is historical in ADL, and will be changed in ADL2 to the proper form of dADL for nested containers, i.e. removing the "items = <" blocks altogether.

8.7.4 Constraint_definitions Section

The `constraint_definitions` section is of exactly the same form as the `term_definitions` 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_bindings` section. An example of a constraint term definition for the hepatitis constraint is as follows:

```
items = <
    ["ac1015"] = <
       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.

8.7.5 Term_bindings Section

This section is used to describe the equivalences between archetype local terms and terms found in external terminologies. The main purpose for allowing query engine software that 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 *open*EHR Reference Model `DV_TEXT` and `DV_CODED_TEXT` types.

Global Term Bindings

There are two types of term bindings that can be used, 'global' and path-based. The former is where an external term is bound directly to an archetype local term, and the binding holds globally throughout the archetype. In many cases, archetype terms only appear once in an archetype, but in some archetypes, at-codes are reused throughout the archetype. In such cases, a global binding asserts that the correspondence is true in all locations. A typical global term binding section resembles the following:

```
term_bindings = <
     ["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 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.

Path-based Bindings

The second kind of binding is one between an archetype path and an external code. This occurs commonly for archetypes where a term us re-used at the leaf level. For example, in the binding example below, the at0004 code represents 'temperature' and the codes at0003, at0005, at0006 etc correspond to various times such as 'any', 1-hour average, 1-hour maximum and so on. Some terminologies (notably LOINC, the laboratory terminology in this example) define 'pre-coordinated' codes, such as '1 hour body temperature'; these clearly correspond not to single codes such as at0004 in the archetype, but to whole paths. In such cases, the key in each term binding row is a full path rather than a single term.

```
["LNC205"] = <
  items = <
     ["/data[at0002]/events[at0003]/data[at0001]/item[at0004]"] = <[LNC205::8310-5]>
     ["/data[at0002]/events[at0005]/data[at0001]/item[at0004]"] = <[LNC205::8321-2]>
     ["/data[at0002]/events[at0006]/data[at0001]/item[at0004]"] = <[LNC205::8311-3]>
     ["/data[at0002]/events[at0007]/data[at0001]/item[at0004]"] = <[LNC205::8316-2]>
     ["/data[at0002]/events[at0008]/data[at0001]/item[at0004]"] = <[LNC205::8332-0]>
     ["/data[at0002]/events[at0009]/data[at0001]/item[at0004]"] = <[LNC205::8312-1]>
     ["/data[at0002]/events[at0017]/data[at0001]/item[at0004]"] = <[LNC205::8325-3]>
```

```
            ["/data[at0002]/events[at0019]/data[at0001]/item[at0004]"] = <[LNC205::8320-4]>
        >
    >
```

8.7.6 Constraint_bindings Section

The last of the ontology sections formally describes bindings to placeholder constraints (see Placeholder Constraints on page 59) 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_bindings = <
        [“snomed_ct”] = <
            items = <
                [“ac0001”] = <http://terminology.org?query_id=12345>
                [“ac0002”] = <http://terminology.org?query_id=678910>
            >
        >
    >
```

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

8.8 Annotations Section

The annotations section of an archetype provides a place for node-level meta-data to be added to the archetype. This can be used during the design phase to track dependencies, design decisions, and specific resource references. Each annotation is keyed by the path of the node being annotated, and may have any number of tagged elements. A typical annotations section looks as follows.

```
    annotations
        annotations = <
            ["/data/items[at0.37]/items[at0.38]/value"] = < -- Clin st. / Stage
                items = <
                    ["message requirement"] = <"staging field in msg type 2345">
                    ["guideline"] = <"http://guidelines.org/gl24.html#staging">
                    ["data dict equivalent"] = <"NHS data item aaa.1">
                >
            >
            ["/data/items[at0.37]/items[at0.39]/value"] = < -- Clin st. / Tumour
                items = <
                    ["message requirement"] = <"tumour field in msg type 2345">
                    ["guideline"] = <"http://guidelines.org/gl24.html#mass">
                    ["data dict equivalent"] = <"NHS data item aaa.2">
                >
            >
        >
```

8.9 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 optional, and 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 = <
        ["1.57"] = <
            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">
        >
        -- etc
        ["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">
        >
    >
```

9 Specialisation

9.1 Overview

Archetypes can be specialised in a similar way to classes in object-oriented programming languages. Common to both situations is the use of a *differential* style of declaration, i.e. the contents of a specialised entity are expressed as differences with respect to the parent - previously defined elements from the parent that are not changed are not repeated in the descendant. Two extra constructs are included in the ADL syntax to support redefinition in specialised archetypes.

The basic test that must be satisfied by a specialised archetype is as follows:

- All possible data instance arrangements that conform to the specialised archetype must also conform to all of its parents, recursively to the ultimate parent.

This condition ensures that data created by a specialised archetype that is not itself shared by two systems can be processed by the use of a more general parent that is shared.

The semantics that allow this are similar to the 'covariant redefinition'[1] notion used in some object-oriented programming languages, and can be summarised as follows.

- A non-specialised (i.e. top-level) archetype defines an instance space that is a subset of the space defined by the class in the reference information model on which the archetype is based.
- A specialised archetype can specialise only one parent archetype, i.e. single inheritance.
- A specialised archetype defines an instance space defining the following elements:
    - unchanged object and attribute constraints *inherited* from the parent archetype;
    - and one or more:
        - *redefined* object constraints, that are proper subsets of the corresponding parent object constraints;
        - *redefined* attribute constraints, that are proper subsets of the corresponding parent attribute constraints;
        - *extensions*, i.e. object constraints added to a container attribute with respect to the corresponding attribute in the parent archetype, but only as allowed by the underlying reference model.
- All elements defined in a parent archetype are either inherited unchanged or redefined in a specialised child.
- Specialised archetypes are expressed *differentially* with respect to the parent, i.e. they do not mention purely inherited elements, only redefinitions and extensions.

TBD_3: Redefinition cannot remove an object constraint, only narrow it to a reduced instance space.

- Extensions always define an additional subset of the instance space defined by the reference model element being extended (i.e. to which the 'new' objects belong). The extension capability allows archetypes to remain extensible without having to know in advance how or if they will be extended.

1. see http://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

The following sections describe the details of specialisation. The term 'object' is used synonymously with 'object constraint' since all elements in ADL are constraints.

9.2 Examples

The examples below provide a basis for understanding most of the semantics discussed in the subsequent sections.

9.2.1 Redefinition for Specialisation

The example shown in FIGURE 8 illustrates redefinition in a specialised archetype. The first text is taken from the definition section of the 'laboratory result' `OBSERVATION` archetype on *open*EHR.org[1], and contains an `ELEMENT` node whose identifier is `[at0013]`, defined as 'panel item' in the archetype ontology (sibling nodes are not shown here). The intention is that the at0013 node be specialised into particular 'panel items' or analytes according to particular types of test result. Accordingly, the at0013 node has occurrences of 0..* and its value is not constrained with respect to the reference model, meaning that the type of the *value* attribute can be any descendant of `DATA_VALUE`.

The second text is a specialised version of the laboratory result archetype, defining 'thyroid function test result'. The redefinitions include:

- a redefinition of the top-level object node identifier `[at0000]`, with the specialised node identifier `[at0000.1]`;
- eight nodes redefining the `[at0013]` node are shown, with overridden node identifiers `[at0013.2]` - `[at0013.9]`;
- reduced occurrences (0..1 in each case);
- redefinition of the *value* attribute of each `ELEMENT` type to `DV_QUANTITY`, shown in expanded form for node `[at0013.2]` (achieved by an inline dADL section; see section 10.1.1 on page 108).

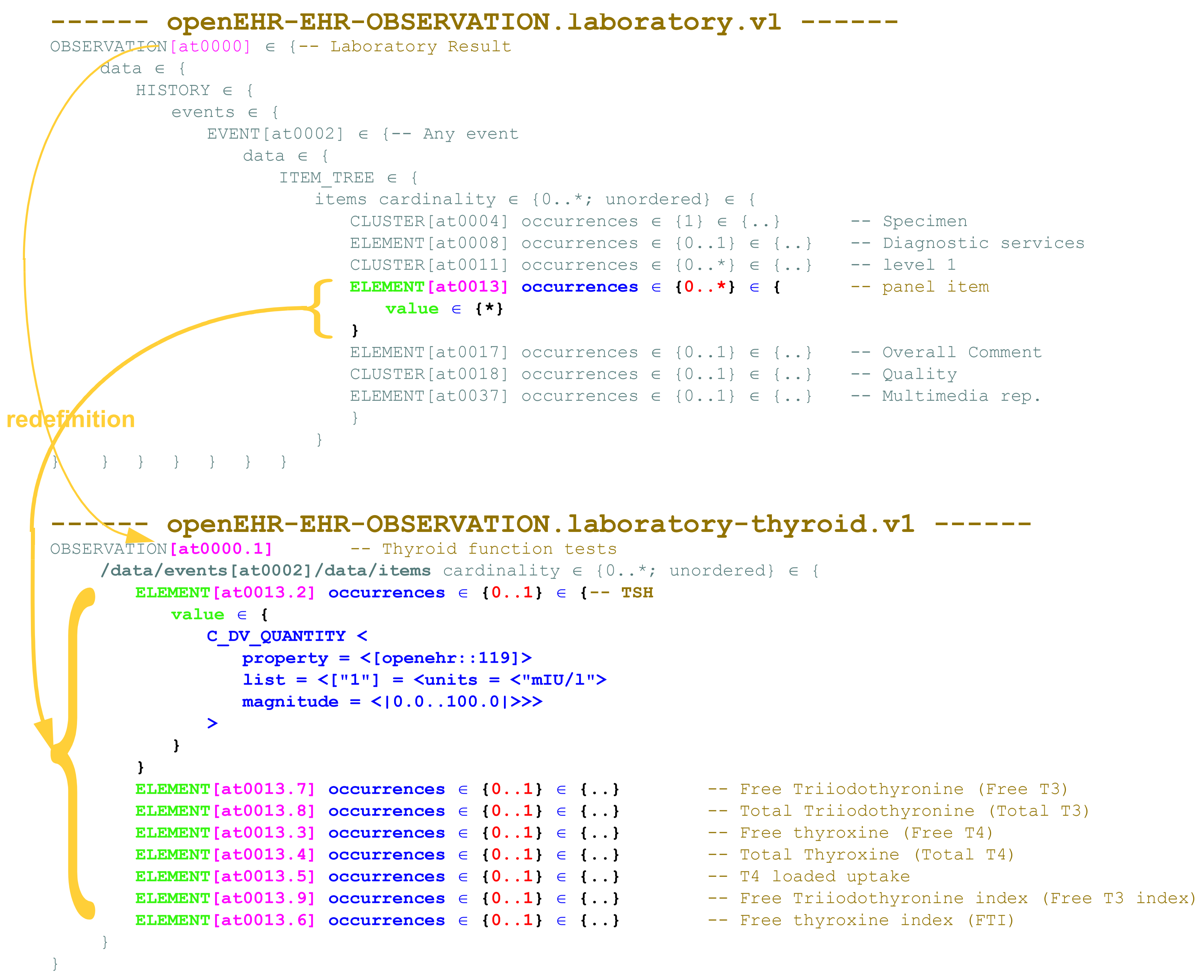
This archetype is typical of a class of specialisations that use only redefinition, due to the fact that all objects in the redefined part of the specialised version are semantically specific kinds of a general object, in this case, 'panel item'.

9.2.2 Redefinition for Refinement

The example shown in FIGURE 9 is taken from the *open*EHR 'Problem' archetype[2] lineage and illustrates the use of redefinition and extension. The first text is the the definition section of the top-level 'Problem' archetype, and shows one `ELEMENT` node in expanded form, with the remaining nodes in an elided form.

The second text is from the 'problem-diagnosis' archetype, i.e. a 'diagnosis' specialisation of the general notion of 'problem'. In this situation, the node `[at0002]`, with occurrences of 1, i.e. mandatory non-multiple, has its meaning narrowed to `[at0002.1]` 'diagnosis' (diagnosed problems are seen as a subset of all problems in medicine), while new sibling nodes are added to the *items* attribute to define details particular to recording a diagnosis. The extension nodes are identified by the codes `[at0.32]`, `[at0.35]` and `[at0.37]`, with the latter two shown in elided form.

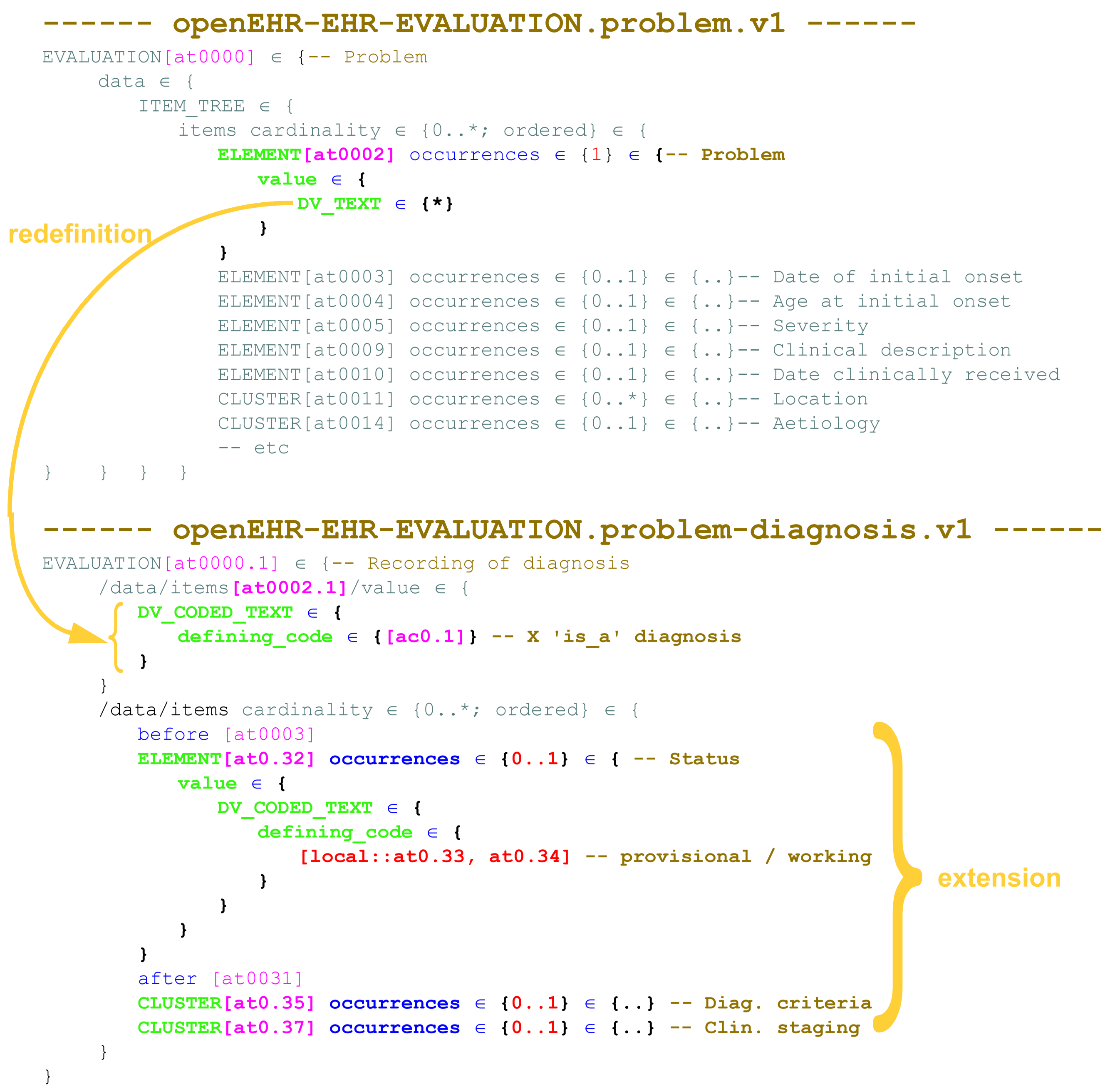
1. http://www.openehr.org/svn/knowledge/archetypes/dev/html/en/openEHR-EHR-OBSERVATION.laboratory.v1.html
2. http://svn.openehr.org/knowledge/archetypes/dev/html/en/openEHR-EHR-EVALUATION.problem.v1.html



```
------ openEHR-EHR-OBSERVATION.laboratory.v1 ------
OBSERVATION[at0000] ∈ {-- Laboratory Result
    data ∈ {
        HISTORY ∈ {
            events ∈ {
                EVENT[at0002] ∈ {-- Any event
                    data ∈ {
                        ITEM_TREE ∈ {
                            items cardinality ∈ {0..*; unordered} ∈ {
                                CLUSTER[at0004] occurrences ∈ {1} ∈ {..}       -- Specimen
                                ELEMENT[at0008] occurrences ∈ {0..1} ∈ {..}    -- Diagnostic services
                                CLUSTER[at0011] occurrences ∈ {0..*} ∈ {..}    -- level 1
                                ELEMENT[at0013] occurrences ∈ {0..*} ∈ {       -- panel item
                                    value ∈ {*}
                                }
                                ELEMENT[at0017] occurrences ∈ {0..1} ∈ {..}    -- Overall Comment
                                CLUSTER[at0018] occurrences ∈ {0..1} ∈ {..}    -- Quality
                                ELEMENT[at0037] occurrences ∈ {0..1} ∈ {..}    -- Multimedia rep.
                            }
                        }
}    }   }   }   }   }   }

------ openEHR-EHR-OBSERVATION.laboratory-thyroid.v1 ------
OBSERVATION[at0000.1]       -- Thyroid function tests
    /data/events[at0002]/data/items cardinality ∈ {0..*; unordered} ∈ {
        ELEMENT[at0013.2] occurrences ∈ {0..1} ∈ {-- TSH
            value ∈ {
                C_DV_QUANTITY <
                    property = <[openehr::119]>
                    list = <["1"] = <units = <"mIU/l">
                    magnitude = <|0.0..100.0|>>>
                >
            }
        }
        ELEMENT[at0013.7] occurrences ∈ {0..1} ∈ {..}        -- Free Triiodothyronine (Free T3)
        ELEMENT[at0013.8] occurrences ∈ {0..1} ∈ {..}        -- Total Triiodothyronine (Total T3)
        ELEMENT[at0013.3] occurrences ∈ {0..1} ∈ {..}        -- Free thyroxine (Free T4)
        ELEMENT[at0013.4] occurrences ∈ {0..1} ∈ {..}        -- Total Thyroxine (Total T4)
        ELEMENT[at0013.5] occurrences ∈ {0..1} ∈ {..}        -- T4 loaded uptake
        ELEMENT[at0013.9] occurrences ∈ {0..1} ∈ {..}        -- Free Triiodothyronine index (Free T3 index)
        ELEMENT[at0013.6] occurrences ∈ {0..1} ∈ {..}        -- Free thyroxine index (FTI)
    }
}
```

FIGURE 8 Specialised archetype showing redefinition to multiple children



```
------ openEHR-EHR-EVALUATION.problem.v1 ------
EVALUATION[at0000] ∈ {-- Problem
     data ∈ {
         ITEM_TREE ∈ {
             items cardinality ∈ {0..*; ordered} ∈ {
                 ELEMENT[at0002] occurrences ∈ {1} ∈ {-- Problem
                     value ∈ {
                         DV_TEXT ∈ {*}
                     }
                 }
                 ELEMENT[at0003] occurrences ∈ {0..1} ∈ {..}-- Date of initial onset
                 ELEMENT[at0004] occurrences ∈ {0..1} ∈ {..}-- Age at initial onset
                 ELEMENT[at0005] occurrences ∈ {0..1} ∈ {..}-- Severity
                 ELEMENT[at0009] occurrences ∈ {0..1} ∈ {..}-- Clinical description
                 ELEMENT[at0010] occurrences ∈ {0..1} ∈ {..}-- Date clinically received
                 CLUSTER[at0011] occurrences ∈ {0..*} ∈ {..}-- Location
                 CLUSTER[at0014] occurrences ∈ {0..1} ∈ {..}-- Aetiology
                 -- etc
}    }   }   }

------ openEHR-EHR-EVALUATION.problem-diagnosis.v1 ------
EVALUATION[at0000.1] ∈ {-- Recording of diagnosis
     /data/items[at0002.1]/value ∈ {
         DV_CODED_TEXT ∈ {
             defining_code ∈ {[ac0.1]} -- X 'is_a' diagnosis
         }
     }
     /data/items cardinality ∈ {0..*; ordered} ∈ {
         before [at0003]
         ELEMENT[at0.32] occurrences ∈ {0..1} ∈ { -- Status
             value ∈ {
                 DV_CODED_TEXT ∈ {
                     defining_code ∈ {
                         [local::at0.33, at0.34] -- provisional / working
                     }
                 }
             }
         }
         after [at0031]
         CLUSTER[at0.35] occurrences ∈ {0..1} ∈ {..} -- Diag. criteria
         CLUSTER[at0.37] occurrences ∈ {0..1} ∈ {..} -- Clin. staging
     }
}
```

FIGURE 9 Specialised archetype showing redefinition and extension

9.3 Specialisation Concepts

9.3.1 Differential and Flat Forms

Specialised archetypes in their authored form are represented in 'differential' form. The syntax is the same as for non-specialised archetypes, with two additions: specialisation paths (see section 9.3.3) and ordering indicators (see Ordering of Sibling Nodes on page 97). For a specialised archetype therefore, the lineage of archetypes back to the ultimate parent must be taken into account in order to obtain its complete semantics.

Differential form means that the only attributes or objects mentioned are those that redefine corresponding elements in the parent and those that introduce new elements. The differential approach to representation of specialised archetypes give rise to the need for a *flat form* of a specialised archetype: the equivalent archetype defined by the sum of the (differential) child and its parent, as if the child archetype had been defined standalone. The flat form of archetypes is used for building templates, and subsequently at runtime. It is generated by 'compressing' the effects of inheritance of the parent to the specialised child into a single archetype, and applies recursively all the way up an *archetype lineage* to the ultimate parent, which must be a top-level (non-specialised) archetype. For a top-level archetype, the flat-form is the same as its differential form (i.e. in a top-level archetype, every node is considered to be an extension node).

9.3.2 Specialisation Levels

In order to talk about archetypes at different levels of specialisation, a standard way of identifying the levels of specialisation is used, as follows:

- level 0: top-level, non-specialised archetypes
- level 1: specialisations of level 0 archetypes
- level 2: specialisations of level 1 archetypes
- etc.

For nodes carrying a node identifier, the specialisation level is always equal to the number of '.' characters found in the identifier.

9.3.3 Specialisation Paths

Because ADL is a block-structured language, the redefinition of nodes deep in the parent structure normally requires descending into the structure. Since it is common to want to further constrain only nodes deep within a structure in specialised archetype, a more convenient way is provided in ADL to do this using a *specialisation path*, illustrated by the following example:

```
OBSERVATION[at0000.1] ∈ {-- Thyroid function tests
    /data/events[at0002]/data/items ∈ {
        ELEMENT[at0013.2] occurrences ∈ {0..1} ∈ {-- TSH
            value ∈ {
                C_DV_QUANTITY <...>
            }
        }
        ELEMENT[at0013.7] occurrences ∈ {0..1} ∈ {..}-- Free T3
        ....
      }
    }
}
```

In this fragment, a path is used rather than an attribute name. A path can be used in this manner only if no further constraints are required 'on the way' into the deep structure.

The rules for specialisation paths are as follows.

- A specialisation path is constructed down to the first attribute having any child objects to be further constrained in the present archetype.
- The shortest useful path that can be used is '/' followed by an attribute name from the top level class being constrained by the archetype.

The paths `/data`, `/state` and `/protocol` would all be valid override paths in an *open*EHR `OBSERVATION` archetype.

9.3.4 Path Congruence

Any node in an archetype can unambiguously be located by its archetype path. For example, the text value of the 'problem' node of the `openEHR-EHR-EVALUATION.problem.v1` archetype shown at the top of FIGURE 9 is:

```
/data/items[at0002]/value
```

Similarly the path to the redefined version of the same node in the `openEHR-EHR-EVALUATION.problem-diagnosis.v1` archetype at the bottom of the same figure is:

```
/data/items[at0002.1]/value
```

By inspection, it can be seen that this path is a variant of the corresponding path in the parent archetype, where a particular object node identifier has been specialised.

In general, the path of every redefined node in a specialised archetype will have a direct equivalent in the parent archetype, which can be determined by removing one level of specialisation from any node identifiers within the specialised path *that are at the level of specialisation of the specialised archetype* (i.e. node identifiers corresponding to higher specialisation levels are not changed). In this way, the nodes in a specialised archetype source can be connected to their counterparts in parent archetypes, for purposes of validation and flattening.

Conversely, any given path in an archetype that has children will have congruent paths in the children wherever nodes have been specialised.

9.3.5 Redefinition Concepts

A specialised archetype definition section is expressed in terms of:

- *redefined* object constraints;
- *redefined* attribute constraints;
- *extensions*, being object constraints added to container attributes.

Apart from *existence* and *cardinality* redefinition, all specialisation relates to object constraints. In the ADL syntax, objects can be specified in two places: under a single-value attributes and under multiply-valued (container) attributes.

Each object under a single-valued attribute defines an alternative that may be used to constrain data at that attribute position. An example is the `OBSERVATION`.*protocol* attribute from the *open*EHR reference model: if multiple objects appear under this attribute, only one can be used at runtime to constrain data.

Within a container attribute, the meaning of multiple objects is that each child object defines constraints on one or more members of the container in the data. The *occurrences* constraint on each one determines how many objects in the data match a given object constraint in the attribute.

Object constraints can be specialised in both places by redefinition, refinement and exclusion. In addition, extension can be used under container attributes. The actual semantics are described in terms of object node identification, type redefinition, and structural constraints (existence, cardinality and occurrences), and are mostly the same for objects under single- and multiply-valued attributes. The following sections describe the details.

9.4 Attribute Redefinition

A small number of things can be redefined on attributes, including existence and cardinality. A basic rule of redefinition is that a specialised archetype cannot change the multiplicity type of an attribute.

9.4.1 Existence

All attributes mentioned in an archetype have an *existence* constraint, indicating whether a value is required or not. The constraint is either stated explicitly - typically done for single-valued attirbutes - or it is the value from the reference model - typical for multiply-valued attributes. In both cases, the existence of an attribute in a parent archetype can be redefined in a specialised archetype using the standard cADL syntax. In the following example, an implicit existence constraint picked up from the reference model of {0..1} is redefined in a child archetype to {1}, i.e. mandatory.

Parent archetype:

```
OBSERVATION[at0000] ∈ { -- blood pressure measurement
    protocol ∈ { -- existence not changed from reference model
       -- etc
    }
}
```

Child archetype:

```
OBSERVATION[at0000.1] ∈ { -- paediatric blood pressure measurement
    protocol existence ∈ {1} ∈ {
       -- etc
    }
}
```

Redefinition of existence to {0}, i.e. prohibited is possible by this method, but is likely to indicate poor design of the archetype, given that the decision to remove optional attributes is much more likely to be local, and therefore more appropriate in templates rather than archetypes.

9.4.2 Container Attributes

The following sub-sections describe specialisation semantics specific to container attributes.

9.4.2.1 Cardinality

The *cardinality* constraint defines how many object instances can be in the container within the data (not the archetype). In a specialised archetype, cardinality can be redefined to be a narrower range than in the parent, further limiting the valid ranges of items in the data that may occur within the container. This would normally only make sense if refinements were made to the occurrences of the contained items, i.e.:

- narrowing the occurrences range of an object;

- excluding an object by setting its occurrences to {0};
- adding new objects, which themselves will have occurrences constraints;
- setting some object occurrences to mandatory, and the enclosing cardinality lower limit to some non-zero value.

As long as the relationship between the enclosing attribute’s cardinality constraint and the occurrences constraints defined on all the contained items (including those inherited unchanged, and therefore not mentioned in the specialised archetype) is respected (see VCOC validity rule, AOM specification), any of the above specialisations can occur.

The following provides an example of cardinality redefinition.

Parent archetype:

```
ITEM_LIST ∈ { -- general check list
    items cardinality ∈ {0..*} ∈ { -- any number of items
       ELEMENT[at0012] occurrences ∈ {0..*} ∈ {} -- generic checklist item
    }
}
```

Child archetype:

```
ITEM_LIST ∈ { -- pre-operative check list
    items cardinality ∈ {3..10} ∈ { -- at least 3 mandatory items
       ELEMENT[at0012.1] occurrences ∈ {1} ∈ {} -- item #1
       ELEMENT[at0012.2] occurrences ∈ {1} ∈ {} -- item #2
       ELEMENT[at0012.3] occurrences ∈ {1} ∈ {} -- item #3
       ELEMENT[at0012.4] occurrences ∈ {0..1} ∈ {} -- item #4
       ...
       ELEMENT[at0012.10] occurrences ∈ {0..1} ∈ {} -- item #10
    }
}
```

9.4.2.2 Ordering of Sibling Nodes

Within container attributes, the order of objects may be significant from the point of view of domain users, i.e. the container may be considered as an ordered list. This is easy to achieve in top-level archetype, using the ‘ordered’ qualifier on a cardinality constraint. However when particular node(s) are redefined into multiple specialised nodes, or new nodes added by extension, the desired order of the new nodes may be such that they should occur interspersed at particular locations among nodes defined in the parent archetype. The following text is a slightly summarised view of the *items* attribute from the problem archetype shown in FIGURE 9:

```
items cardinality ∈ {0..*; ordered} ∈ {
    ELEMENT[at0002] occurrences ∈ {1} ∈ {..}     -- Problem
    ELEMENT[at0003] occurrences ∈ {0..1} ∈ {..} -- Date of initial onset
    ELEMENT[at0004] occurrences ∈ {0..1} ∈ {..} -- Age at initial onset
    ELEMENT[at0005] occurrences ∈ {0..1} ∈ {..} -- Severity
    ELEMENT[at0009] occurrences ∈ {0..1} ∈ {..}  -- Clinical description
    ELEMENT[at0010] occurrences ∈ {0..1} ∈ {..}  -- Date clinically received
    CLUSTER[at0011] occurrences ∈ {0..*} ∈ {..}  -- Location
    CLUSTER[at0014] occurrences ∈ {0..1} ∈ {..}  -- Aetiology
    CLUSTER[at0018] occurrences ∈ {0..1} ∈ {..}  -- Occurrences or exacerb’ns
    CLUSTER[at0026] occurrences ∈ {0..1} ∈ {..}  -- Related problems
    ELEMENT[at0030] occurrences ∈ {0..1} ∈ {..}  -- Date of resolution
    ELEMENT[at0031] occurrences ∈ {0..1} ∈ {..}  -- Age at resolution
}
```

To indicate significant ordering in the specialised problem-diagnosis archetype, the keywords `before` and `after` can be used, as follows:

```
    /data/items ∈ {
        before [at0003]
        ELEMENT[at0002.1] ∈ {..}                       -- Diagnosis
        ELEMENT[at0.32] occurrences ∈ {0..1} ∈ {..}  -- Status
        after [at0026]
        CLUSTER[at0.35] occurrences ∈ {0..1} ∈ {..} -- Diagnostic criteria
        CLUSTER[at0.37] occurrences ∈ {0..1} ∈ {..} -- Clinical Staging
    }
```

These keywords are followed by a node identifier reference, and act to modify the node definition immediately following. Technically the following visual rendition would be more faithful, but it is less readable, and makes no difference to a parser:

```
        after [at0026] CLUSTER[at0.35] occurrences ∈ {0..1} ∈ {..} -- etc
```

The rules for specifying ordering are as follows.

- Ordering is only applicable to object nodes defined within a multiply-valued attribute whose cardinality includes the `ordered` constraint;
- Any `before` or `after` statement can refer to the node identifier of any sibling node known in the flat form of the archetype, i.e.:
  - the identifier of any redefined node;
  - the identifier of any new node;
  - the identifier of any inherited node that is not redefined amongst the sibling nodes.
- If no ordering indications are given, redefined nodes should appear in the same position as the nodes they replace, while extension nodes should appear at the end.

If ordering indicators are used in an archetype that is itself further specialised, the following rules apply:

- If the referenced identifier becomes unavailable due to being redefined in the new archetype, it must be redefined to refer to an available sibling identifier as per the rules above.
- If this does not occur, a `before` reference will default to the *first* sibling node identifier currently available conforming to the original identifier, while an after reference will default to the *last* such identifier available in the current flat archetype.

TBD_4: should we also introduce 'first' and 'last' keywords to allow new nodes to always be forced to top or bottom of the list, regardless of what else is there?

TBD_5: what happens if, due to multiple levels of redefinition, there is > 1 candidate to go before a given node?

9.5 Object Redefinition

Object redefinition can occur for any object constraint in the parent archeype, and can include redefinition of node identifier, occurrences, reference model type. For certain kinds of object constraints, specific kinds of redefinition are possible.

9.5.1 Node Identifiers

In an archetype, node identifiers ('at-codes') are mandatory on all object constraints defined as children of a multiply-valued attribute and multiple same-typed children of single-valued attributes (see Node Identifiers on page 47). They are optional on other single child constraints of single-valued attributes. This rule applies in specialised as well as top-level archetypes.

Redefinition semantics for node identifiers are somewhat special. They can be redefined for purely semantic purposes (e.g. to redefine 'heart rate' to 'fetal heart rate') - this is what happens on the root node of every archetype. However, if any other aspect of the immediate object node, i.e. occurrences, reference model type, or the constraint type is changed, the node identifier *must be redefined*.

The obvious alternative would have been to require that every identified node from the point of redefinition (or extension or exclusion) to the top of the structure be deemed as redefined, and therefore carry a specialised node identifier, where appropriate. This approach is not used for two reasons. Firstly, the presence of any specialised node in the hierarchy from a leaf node to the root creates a specialised path particular to the leaf node, which can be used to identify data created by the specialised archetype. Secondly, the meaning of nodes above the changed node do not change as such, and creating specialised node identifiers would require the definition of superfluous codes in the archetype ontology.

9.5.1.1 Redefined Nodes

Where an identifier is defined on a node in a parent archetype, redefined versions of the node carry a node identifier that is a specialised version of the identifier of the original. As a consequence, this means that the identifier of the root node of a specialised archetype will always be redefined, indicating the more specific 'type' of instances it defines. The first example above defines a 'thyroid function test' `[at0000]` subset of 'laboratory result' `[at0000.1]` `OBSERVATION` objects.

Specific redefinitions and extensions occur at lower points in the structure. In the first example above, the node `[at0013]` is redefined into a number of more specialised nodes `[at0013.2]` - `[at0013.9]`, while in the second, the identifier `[at0002]` is redefined to a single node `[at0002.1]`.

The syntactic form of the identifier of a redefined node is a copy of the original followed by a dot ('.'), optionally intervening instances of the pattern '0.' and then a further non-zero number, i.e.:

- atNNNN {.0}* .N

This permits node identifiers from a given level to be redefined not just at the next level, but at multiple levels below.

Examples of redefined node identifiers:

- at0001.1 -- redefinition of at0001 at level 1 specialisation
- at0001.0.1 -- redefinition of at0001 node in level 2 specialisation archetype
- at0001.1.1 -- redefinition of at0001.1 in level 2 specialisation archetype.

Redefined versions of nodes with no node id in the parent archetype do not require a node identifier in the child archetype.

9.5.1.2 Extension Nodes

Extension nodes carry node identifiers according to the rule mentioned above. The second example includes the new node identifiers `[at0.32]`, `[at0.35]` and `[at0.37]`, whose codes start with a '0'. indicating that they have no equivalent code in the parent archetype.

The node identifier syntax of an extension node commences with at least one instance of the pattern '0.'. The structure of node identifiers for both kinds of node thus always indicates at what level the identifier was introduced, given by the number of dots.

Examples of redefined node identifiers:

- at0.1 -- identifier of extension node introduced at level 1
- at0.0.1 -- identifier of extension node introduced at level 2

When a flat form is created, the level at which any given node was introduced or redefined is clear due to the identifier coding system.

TBD_6: to be properly formal, all containers should be declared open or closed in archetypes; only open ones should be extensible. Currently we are assuming that all containers are 'open'.

9.5.2 Occurrences Redefinition and Exclusion

The `occurrences` constraint on an object constraint indicates how many instances within the data may conform to that constraint (see Container Attributes on page 50). Within container attributes, `occurrences` is usually redefined in order to make a given object mandatory rather than optional; it can also be used to exclude an object constraint. In the following example, the occurrences of the `[at0003]` node is redefined from {0..1} i.e. optional, to {1}, i.e. mandatory.

Parent (`openEHR-EHR-EVALUATION.problem.v1`):

```
EVALUATION[at0000] ∈ {-- Problem
    data ∈ {
        ITEM_TREE ∈ {
            items cardinality ∈ {0..*; ordered} ∈ {
                ELEMENT[at0002] occurrences ∈ {1} ∈ {..}-- Problem
                ELEMENT[at0003] occurrences ∈ {0..1} ∈ ∈ {..}-- Date of initial onset
                -- etc
}    }    }    }
```

Child (`openEHR-EHR-EVALUATION.problem-diagnosis.v1`):

```
EVALUATION[at0000.1] ∈ {-- Recording of diagnosis
    data ∈ {
        ITEM_TREE ∈ {
            items cardinality ∈ {0..*; ordered} ∈ {
                ELEMENT[at0002] occurrences ∈ {1} ∈ {..}-- Problem
                ELEMENT[at0003] occurrences ∈ {1} ∈ ∈ {..}-- Date of initial onset
                -- etc
}    }    }    }
```

Occurrences is normally only constrained on child objects of container attributes, but can be constrained on objects of any attribute to effect exclusion of part of the instance space. This can be useful where a number of alternatives for a single-valued attribute have been stated, and the need is to remove some alternatives in a specialised child archetype. For example, an archetype might have the following constraint:

```
ELEMENT[at0003] ∈ {
    value ∈ {
        DV_QUANTITY ∈ {*}
        DV_INTERVAL<DV_QUANTITY> ∈ {*}
        DV_COUNT ∈ {*}
        DV_INTERVAL<DV_COUNT> ∈ {*}
    }
}
```

and the intention is to remove the `DV_INTERVAL<*>` alternatives. This is achieved by redefining the enclosing object to removed the relevant types:

```
ELEMENT[at0003.1] ∈ {
    value ∈ {
        DV_INTERVAL<DV_QUANTITY> occurrences ∈ {0}
        DV_INTERVAL<DV_COUNT> occurrences ∈ {0}
    }
}
```

9.5.3 Reference Model Type Refinement

The type of an object may be redefined to one of its subtypes as defined by the reference model. A typical example of where this occurs in archetypes based on the *open*EHR reference model is when ELEMENT.*value* is constrained to '*' in a parent archetype, meaning 'no further constraint on its RM type of DATA_VALUE', but is then constrained in a specialised archetype to subtypes of DATA_VALUE, e.g. DV_QUANTITY or DV_PROPORTION[1]. The following figure containts a simplified extract of the data values part of the *open*EHR reference model, and is the basis for the examples below.

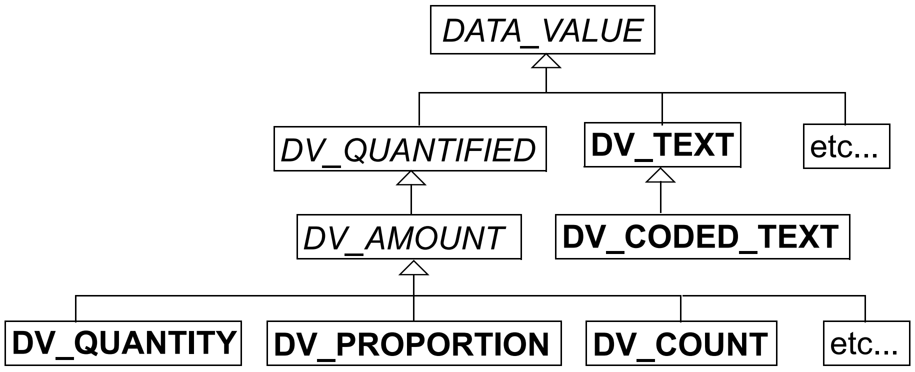


FIGURE 10 Example reference model type structure

The most basic form of type refinement is shown in the following example:

Parent archetype:

```
    value ∈ {*} -- any subtype of DATA_VALUE, from the ref model
```

Specialised archetype:

```
    .../value ∈ {
        DV_QUANTITY ∈ {*} -- now limit to the DV_QUANTITY subtype
    }
```

The meaning of the above is that instance data constrained by the specialised archetype at the value node must match the DV_QUANTITY constraint only - no other subtype of DATA_VALUE is allowed.

When a type in an archetype is redefined into one of its subtypes, any existing constraints on the original type in the parent archetype are respected. In the following example, a DV_AMOUNT constraint that required *accuracy* to be present and in the range +/-5% is refined into a DV_QUANTITY in which two attributes of the subtype are constrained. The original *accuracy* attribute is inherited without change.

Parent archetype:

```
    value ∈ {
        DV_AMOUNT ∈ {
            accuracy ∈ {|-0.05..0.05|}
        }
    }
```

Specialised archetype:

```
    .../value ∈ {
        DV_QUANTITY ∈ {
            magnitude ∈ {|2.0..10.0|}
```

1. See the *open*EHR data types specification at http://www.openehr.org/releases/1.0.1/architecture/rm/data_types_im.pdf for details.

```
            units ∈ {"mmol/ml"}
        }
    }
```

In the same manner, an object node can be specialised into more than one subtype, where each such constraint selects a mutually exclusive subset of the instance space. The following example shows a specialisation of the DV_AMOUNT constraint above into two subtyped constraints.

```
    .../value ∈ {
        DV_QUANTITY ∈ {
            magnitude ∈ {|2.0..10.0|}
            units ∈ {"mmol/ml"}
        }
        DV_PROPORTION ∈ {
            numerator ∈ {|2.0..10.0|}
            type ∈ {pk_unitary}
        }
    }
```

Here, instance data may only be of type DV_QUANTITY or DV_PROPORTION, and must satisfy the respective constraints for those types.

A final variant of subtyping is when the intention is to constraint the data to a supertype with exceptions for particular subtypes. In this case, constraints based on subtypes are matched first, with the constraint based on the parent type being used to constrain all other subtypes. The following example constrains data at the *value* node to be:

- an instance of DV_QUANTITY with *magnitude* within the given range etc;
- an instance of DV_PROPORTION with *numerator* in the given range etc;
- an instance of any other subtype of DV_AMOUNT, with *accuracy* in the given range.

```
    .../value ∈ {
        DV_QUANTITY ∈ {
            magnitude ∈ {|2.0..10.0|}
            units ∈ {"mmol/ml"}
        }
        DV_PROPORTION ∈ {
            numerator ∈ {|2.0..10.0|}
            type ∈ {pk_unitary}
        }
        DV_AMOUNT ∈ {
            accuracy ∈ {|-0.05..0.05|}
        }
    }
```

A typical use of this kind of refinement in *open*EHR would be to add an alternative for a DV_CODED_TEXT constraint for a specific terminology to an existing DV_TEXT constraint in a *name* attribute, as follows:

```
    name ∈ {
        DV_CODED_TEXT ∈ {
            defining_code ∈ {[Snomed_ct::]}
        }
        DV_TEXT ∈ {
            value ∈ {/.+/} -- non-empty string
        }
    }
```

All of the above specialisation based on reference model subtypes can be applied in the same way to identified object constraints.

9.5.4 Placeholder Constraint Redefinition

A placeholder constraint is used to specify a constraint whose actual value range is defined externally from the archetype, as shown in the following example.

```
ELEMENT [at0013] ∈ { -- cuff size
    value ∈ {
        DV_CODED_TEXT ∈ {
            defining_code ∈ {
                CODE_PHRASE ∈ {[ac0001]}
            }
        }
    }
}
```

In a specialisation of the same archetype, the placeholder constraint has been replaced by a constraint of the same reference model type (`CODE_PHRASE`), which happens to have its own ADL syntax defined in the *open*EHR Archetype Profile.

```
ELEMENT [at0013] ∈ { -- cuff size
    value ∈ {
        DV_CODED_TEXT ∈ {
            defining_code ∈ {
                [local::
                at0022, -- child cuff
                at0023] -- infant cuff
            }
        }
    }
}
```

The redefinition is assumed to be legal, although it is not directly validatable unless the ac-code in the parent is bound to an actual terminology subset which is available from an appropriate service.

9.5.5 Internal Reference Redefinition

An archetype internal reference, or `use_node` constraint is used to refer to a previously defined constraint from a point later in the archetype. These references can be redefined in two ways, as follows.

- *Target redefinition*: the target constraint of reference may be itself redefined. The meaning for this is that all internal references now assumed the redefined form.
- *Reference redefinition*: one or more `use_node` reference points in the specialised archetypes can be redefined into a concrete constraint that a) replaces the reference, and b) must be completely conformant to the structure which is the target of the original reference.

Two further possible types of redefinition are possible, but seem of limited use, and for the moment are not considered formally part of ADL.

- *Target-only redefinition*: if the intention is to redefine a structure referred to by `use_node` constraints, but to leave the constraints at the reference source points in the original form, each `use_node` reference needs to be redefined by a copy of its target structure, prior to redefinition of the structure at the target location.
- *Reference re-targetting*: an internal reference could potentially be redefined into a reference to a different target whose structure conforms to the original target.
- *Reference occurrences redefinition*: an internal reference may already carry its own occurrences, distinct from the occurrences of the object at the target point. Both occurrences at the

reference source and target points can be redefined in the normal way. Note that if the target of a reference is redefined to have occurrences of {0}, the archetype tooling should warn the user of the effect on any references, since by default they will now refer to an excluded object.

The following validity rules apply to internal references:

`To Be Continued:`

9.5.6 Slot Redefinition

A slot can be redefined so that the archetypes that match the new constraint constitute a reduced sub-set of those matching the original constraint. No special syntax is required - the same syntax as used in the parent archteype is used. Pure exclusions can be used to reduce the matching set of archetypes.

`To Be Continued:`

9.5.7 Primitive Object Redefinition

For terminal objects (i.e. elements of the type `C_PRIMITIVE_OBJECT`) redefinition consists of:

- redefined value ranges or sets using a narrower value range or set;
- exclusions on the previously defined value ranges or sets which have the effect of narrowing the original range or set.

The following example shows a redefined real value range.

Parent archetype:

```
value ∈ {
    DV_QUANTITY ∈ {
        magnitude ∈ {|2.0..10.0|}
        units ∈ {“mmol/ml”}
    }
}
```

Specialised archetype:

```
.../value ∈ {
    DV_QUANTITY ∈ {
        magnitude ∈ {|4.0..6.5|}
    }
}
```

The following example shows a redefined `CODE_PHRASE` value set.

Parent archetype:

```
ELEMENT[at0007] occurrences ∈ {0..*} ∈ {-- System
    name ∈ {
        DV_CODED_TEXT ∈ {
            defining_code ∈ {
                [local::
                at0008, -- Cardiovascular system
                at0009, -- Respiratory system
                at0010, -- Gastro-intestinal system
                at0011, -- Reticulo-Endothelial system
                at0012, -- Genito-urinary system
                at0013, -- Endocrine System
                at0014, -- Central nervous system
                at0015] -- Musculoskeletal system
            }
```

```
                }
            }
        }
```

Specialised archetype:

```
        .../name/defining_code ∈ {
            [local::
            at0010, -- Gastro-intestinal system
            at0011, -- Reticulo-Endothelial system
            at0012, -- Genito-urinary system
            at0013, -- Endocrine System
            at0015] -- Musculoskeletal system
        }
```

In the following example, the exclusion operator ∉ (text form: 'not matches') is used to remove particular values from a value set.

Parent archetype:

```
        ELEMENT[at0007] occurrences ∈ {0..*} ∈ {-- System
            name ∈ {
                DV_CODED_TEXT ∈ {
                    defining_code ∈ {
                        [local::
                        at0008, -- Cardiovascular system
                        at0009, -- Respiratory system
                        at0010, -- Gastro-intestinal system
                        at0011, -- Reticulo-Endothelial system
                        at0012, -- Genito-urinary system
                        at0013, -- Endocrine System
                        at0014, -- Central nervous system
                        at0015] -- Musculoskeletal system
                    }
                }
            }
        }
```

Specialised archetype:

```
        .../name/defining_code ∉ {
            [local::
            at0012, -- Genito-urinary system
            at0013] -- Endocrine System
        }
```

The same kind of statement can be used to exclude values from integer ranges, lists, date patterns and all other primitive types, and also on all `C_DOMAIN_TYPE` instances (which appear as dADL text sections in ADL).

To Be Continued:

9.6 Invariants

Assertions in archetypes have the effect of further reducing the instance space that conforms to an archetype by specifying relationships between values that must hold. For example the main part of an archetype may specify that two nodes are of type `DV_QUANTITY` and have values greater than 0. An assertion may be added that requires the two values to be related by a fixed factor, for instance if they record weight in pounds and kilograms. This has the effect of reducing the set of instances that conform to only those where this relation holds.

In specialised archetypes, further invariants can be added, but existing ones cannot be changed. New invariants cannot logically contradict existing invariants.

To Be Continued:

10 Customising ADL

10.1 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 that is isomorphic to the aggregation structure of the corresponding classes in the reference model. Constraints at the leaf nodes are represented in a syntactic way that avoids committing to particular modelling details. The overall result enables constraints on most reference model types to be expressed. This section describes how to handle exceptions from the standard semantics. *open*EHR uses a small number of such exceptions, which are documented in the *open*EHR Archetype Profile (oAP) document.

A situation in which standard ADL falls short is when the required semantics of constraint are different from those available naturally from the standard approach. Consider a reference model type `QTY`, shown in FIGURE 11, which could be used to represent a person's age in an archetype.

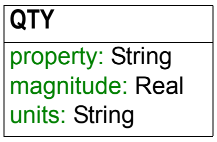


FIGURE 11 Example reference model type

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

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

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

```
age matches {
    QTY matches {
        property matches {"time"}
        units matches {"yr"}
        magnitude matches {|0.0..200.0|}
    }
    QTY matches {
        property matches {"time"}
        units matches {"mth"}
        magnitude matches {|3.0..12.0|}
    }
}
```

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

A more convenient possibility is to introduce a new class into the archetype model, representing the concept "constraint on `QTY`", which we will call `C_QTY`. Such a class fits into the class model of archetypes (see the *open*EHR AOM) by inheriting from the class `C_DOMAIN_TYPE`. A `C_QTY` class is illustrated in FIGURE 12, and corresponds to the way constraints on QTY objects are often expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

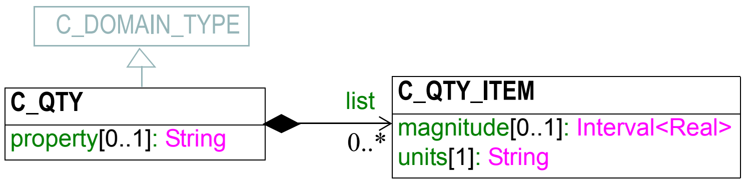


FIGURE 12 Example C_XX Type

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

```
value matches {
    C_QTY <
       property = <“time”>
       list = <
          ["1"] = <
             units = <"yr">
             magnitude = <|0.0..200.0|>
          >
          ["2"] = <
             units = <"mth">
             magnitude = <|1.0..36.0|>
          >
       >
    >
}
```

This approach can be used for any custom type which represents a constraint on a reference model type. Since the syntax is generic, only one change is needed to an ADL parser to support dADL sections within the cADL (definition) part of an archetype. The syntax rules are as follows:

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

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

[Note: that the classes in the above example are a simplified form of classes found in the *open*EHR reference model, designed purely for the purpose of the example.]

10.1.1 Custom Syntax

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

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

However, as with QUANTITY, the most typical constraint required on a CODE_PHRASE is factored differently from the standard ADL - the need is almost always to specify the terminology, and then a set of *code_strings*. A type C_CODE_PHRASE type can be defined as shown in FIGURE 13.

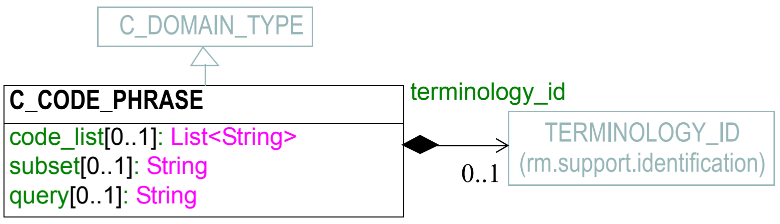


FIGURE 13 C_CODE_PHRASE

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

```
defining_code matches {
    C_CODE_PHRASE <
       terminology_id = <
          value = <"local">
       >
       code_list = <
          ["1"] = <"at0039">
          ["2"] = <"at0040">
       >
    >
}
```

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

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

The above syntax should be understood as an extension to the ADL grammar, and an archetype tool supporting the extension needs to have a modified parser. While these two ADL fragments express exactly the same constraint, the second is shorter and clearer.

Appendix A Relationship of ADL to Other Formalisms

A.1 Overview

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 has been designed for the same use, and none easily express ADL's semantics. During ADL's initial development, 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.

A.2 Constraint Syntaxes

A.2.1 OMG OCL (Object Constraint Language)

The OMG's Object Constraint Language (OCL) appears at first glance 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, if used in the normal way to write constraints on a class model, 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'. This is because doing so would constrain all instances of the class `Company` to conform to the specific configuration of instances corresponding to actors and jugglers, when what is intended is to allow a myriad of possibilities. ADL provides the ability to create numerous archetypes, each describing in detail a concrete configuration of instances of type `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.

A.3 Ontology Formalisms

A.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 [13], 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 description logic (DL), 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 were built based on any particular class model - they 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 as 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 is in terms of constraints on information structures (based on an underlying information model), and the inferencing is between data and the constraints. Some of the differences between ADL and OWL are as follows.

- 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 can 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; Xpath can presumably be used with the RDF representation, although it is not yet clear how meaningful the paths would be with respect to the named categories within an OWL ontology.
- 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.

- ADL provides a rich set of constraints on primitive types, including dates and times. OWL 1.0 (c 2005) did not provide any equivalents; OWL 1.1 (c 2007) look as though it provides some.

Research to date shows that the semantics of an archetype are likely to be representable inside OWL, assuming expected changes to improve its primitive constraint types occur. 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, Spain, Denmark and Turkey(mid 2004).

A.3.2 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 [10]) 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. However, 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, the existing information model and relevant terminologies would have to be converted to KIF statements, before archetypes themselves could be expressed. This is essentially the same process as for expressing archetypes in OWL.

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 [11].

A.4 XML-based Formalisms

A.4.1 XML-schema

Previously, archetypes have been expressed as XML instance documents 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 information model-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.

Appendix B Syntax Specifications

B.1 dADL Syntax

The grammar and lexical specification for the standard dADL syntax is shown below. This grammar is implemented using lex (.l file) and yacc (.y file) specifications for in the Eiffel programming environment. The current release of these files is available at http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/libraries/common_libs/src/structures/syntax/dadl/parser/. The .l and .y files can easily be converted for use in another yacc/lex-based programming environment. The dADL production rules is available as an HTML document.

B.1.1 Grammar

The following provides the dADL parser production rules (yacc specification) as of revision 166 of the Eiffel reference implementation repository (http://www.openehr.org/svn/ref_impl_eiffel).

```
input:
  attr_vals
| complex_object_block
| error

attr_vals:
  attr_val
| attr_vals attr_val
| attr_vals ; attr_val

attr_val:
  attr_id SYM_EQ object_block

attr_id:
  V_ATTRIBUTE_IDENTIFIER
| V_ATTRIBUTE_IDENTIFIER error

object_block:
  complex_object_block
| primitive_object_block
| plugin_object_block

plugin_object_block:
  V_PLUGIN_SYNTAX_TYPE V_PLUGIN_BLOCK

complex_object_block:
  single_attr_object_block
| multiple_attr_object_block

multiple_attr_object_block:
  untyped_multiple_attr_object_block
| type_identifier untyped_multiple_attr_object_block

untyped_multiple_attr_object_block:
  multiple_attr_object_block_head keyed_objects SYM_END_DBLOCK

multiple_attr_object_block_head:
  SYM_START_DBLOCK
```

```
keyed_objects:
  keyed_object
| keyed_objects keyed_object

keyed_object:
  object_key SYM_EQ object_block

object_key:
  [ simple_value ]

single_attr_object_block:
  untyped_single_attr_object_block
| type_identifier untyped_single_attr_object_block

untyped_single_attr_object_block:
  single_attr_object_complex_head SYM_END_DBLOCK
| single_attr_object_complex_head attr_vals SYM_END_DBLOCK

single_attr_object_complex_head:
  SYM_START_DBLOCK

primitive_object_block:
  untyped_primitive_object_block
| type_identifier untyped_primitive_object_block

untyped_primitive_object_block:
  SYM_START_DBLOCK primitive_object_value SYM_END_DBLOCK

primitive_object_value:
  simple_value
| simple_list_value
| simple_interval_value
| term_code
| term_code_list_value

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

type_identifier:
    '(' V_TYPE_IDENTIFIER ')'
  | '(' V_GENERIC_TYPE_IDENTIFIER ')'
  | V_TYPE_IDENTIFIER
  | V_GENERIC_TYPE_IDENTIFIER

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_GT integer_value SYM_ELLIPSIS integer_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM integer_value SYM_ELLIPSIS SYM_LT integer_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM SYM_GT integer_value SYM_ELLIPSIS SYM_LT 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
  | SYM_INTERVAL_DELIM 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_GT real_value SYM_ELLIPSIS real_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM real_value SYM_ELLIPSIS SYM_LT real_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT real_value SYM_ELLIPSIS SYM_LT 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
| SYM_INTERVAL_DELIM 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_ISO8601_EXTENDED_DATE

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_GT date_value SYM_ELLIPSIS date_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM date_value SYM_ELLIPSIS SYM_LT date_value
SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT date_value SYM_ELLIPSIS SYM_LT 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
| SYM_INTERVAL_DELIM date_value SYM_INTERVAL_DELIM

time_value:
  V_ISO8601_EXTENDED_TIME

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_GT time_value SYM_ELLIPSIS time_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM time_value SYM_ELLIPSIS SYM_LT time_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM SYM_GT time_value SYM_ELLIPSIS SYM_LT 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
  | SYM_INTERVAL_DELIM time_value SYM_INTERVAL_DELIM
```

date_time_value:

```
    V_ISO8601_EXTENDED_DATE_TIME
```

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_GT date_time_value SYM_ELLIPSIS date_time_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM date_time_value SYM_ELLIPSIS SYM_LT date_time_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM SYM_GT date_time_value SYM_ELLIPSIS SYM_LT
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
  | SYM_INTERVAL_DELIM date_time_value SYM_INTERVAL_DELIM
```

duration_value:

```
    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_GT duration_value SYM_ELLIPSIS duration_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS SYM_LT duration_value
SYM_INTERVAL_DELIM
  | SYM_INTERVAL_DELIM SYM_GT duration_value SYM_ELLIPSIS SYM_LT
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
  | SYM_INTERVAL_DELIM 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

uri_value:
    V_URI
```

B.1.2 Symbols

The following provides the dADL lexical analyser production rules (lex specification) as of revision 36 of the Eiffel reference implementation repository (http://www.openehr.org/svn/ref_impl_eiffel):

```
----------/* definitions */ -----------------------------------------------
ALPHANUM [a-zA-Z0-9]
IDCHAR [a-zA-Z0-9_]
NAMECHAR [a-zA-Z0-9._\-]
NAMECHAR_SPACE [a-zA-Z0-9._\- ]
NAMECHAR_PAREN [a-zA-Z0-9._\-()]

UTF8CHAR (([\xC2-\xDF][\x80-\xBF])|(\xE0[\xA0-\xBF][\x80-\xBF])|([\xE1-
\xEF][\x80-\xBF][\x80-\xBF])|(\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])|([\xF1-
\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]))

----------/** Separators **/----------------------------------------------

[ \t\r]+            -- Ignore separators
\n+                 -- (increment line count)

----------/** comments **/------------------------------------------------

"--".*              -- Ignore comments
"--".*\n[ \t\r]*    -- (increment line count)

----------/* 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
"??"       -- -> SYM_DT_UNKNOWN
"?"        -- -> Question_mark_code

"|"        -- -> SYM_INTERVAL_DELIM
```

```
"["         -- -> Left_bracket_code
"]"         -- -> Right_bracket_code

"="         -- -> SYM_EQ

">="        -- -> SYM_GE
"<="        -- -> SYM_LE

"<"         -- -> SYM_LT or SYM_START_DBLOCK
">"         -- -> SYM_GT or 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

----------/* V_URI */ -------------------------------------------------
[a-z]+:\/\/[^<>|\\{}^~"\[\] ]*

----------/* V_QUALIFIED_TERM_CODE_REF form [ICD10AM(1998)::F23] */ -----
\[{NAMECHAR_PAREN}+::{NAMECHAR}+\]

----------/* ERR_V_QUALIFIED_TERM_CODE_REF */ -----
\[{NAMECHAR_PAREN}+::{NAMECHAR_SPACE}+\]

----------/* V_LOCAL_TERM_CODE_REF */ ---------------------------------
\[{ALPHANUM}{NAMECHAR}*\]

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

----------/* V_ISO8601_EXTENDED_DATE_TIME YYYY-MM-DDThh:mm:ss[,sss][Z|+/-
nnnn] */ ---
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-9]+)?(Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9](Z|[+-][0-9]{4})?

----------/* V_ISO8601_EXTENDED_TIME hh:mm:ss[,sss][Z|+/-nnnn] */ --------
[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-9]+)?(Z|[+-][0-9]{4})? |
[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})?

----------/* V_ISO8601_EXTENDED_DATE YYYY-MM-DD */ ------------------------
[0-9]{4}-[0-1][0-9]-[0-3][0-9]  |
[0-9]{4}-[0-1][0-9]

----------/* V_ISO8601_DURATION PnYnMnWnDTnnHnnMnn.nnnS */ -------------
-- here we allow a deviation from the standard to allow weeks to be
-- mixed in with the rest since this commonly occurs in medicine

P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?T([0-9]+[hH])?([0-9]+[mM])?([0-9]+(\.[0-
9]+)?[sS])? |
P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?

----------/* V_TYPE_IDENTIFIER */ -------------------------------------
[A-Z]{IDCHAR}*

----------/* V_GENERIC_TYPE_IDENTIFIER */ -------------------------------
[A-Z]{IDCHAR}*<[a-zA-Z0-9,_<>]+>

----------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
[a-z]{IDCHAR}*

----------/* CADL Blocks */ -------------------------------------------
\{[^{}]*                    -- beginning of CADL block
<IN_CADL_BLOCK>\{[^{}]*     -- got an open brace
<IN_CADL_BLOCK>[^{}]*\}     -- got a close brace

----------/* V_INTEGER */ ----------------------------------------------
[0-9]+  |
[0-9]+[eE][+-]?[0-9]+

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

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

\"[^\\\n"]*{                -- beginning of a multi-line string
<IN_STR> {
    \\\\                    -- match escaped backslash, i.e. \\ -> \
    \\\"                    -- match escaped double quote, i.e. \” -> “
    {UTF8CHAR}+             -- match UTF8 chars
    [^\\\n"]+               -- match any other characters
    \\\n[ \t\r]*            -- match LF in line
    [^\\\n"]*\"             -- match final end of string

    .|\n      |
    <<EOF>>                 -- unclosed String -> ERR_STRING
}

----------/* V_CHARACTER */ -------------------------------------------
\'[^\\\n']\'     -- normal character in 0-127
\'\\n\           -- \n
\'\\r\           -- \r
\'\\t\           -- \t
\'\\'\           -- \'
\'\\\\           -- \\
\'{UTF8CHAR}\'   -- UTF8 char
\'.{1,2}   |
\'\\[0-9]+(\/)? -- invalid character -> ERR_CHARACTER
```

B.2 cADL Syntax

The grammar for the standard cADL syntax is shown below. The form used in *open*EHR is the same as this, but with custom additions, described in the *open*EHR Archetype Profile. The resulting grammar and lexical analysis specification used in the *open*EHR reference ADL parser is implemented using lex (.l file) and yacc (.y file) specifications for the Eiffel programming environment. The current release of these files is available at http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/components/adl_parser/src/syntax/cadl/parser. The .l and .y files can be converted for use in other yacc/lex-based programming environments. The production rules of the .y file are available as an HTML document.

B.2.1 Grammar

The following is an extract of the cADL parser production rules (yacc specification) for both flat and differential forms of archetype, as of revision 695 of the Eiffel reference implementation repository (http://www.openehr.org/svn/ref_impl_eiffel). Note that because of interdependencies with path and assertion production rules, practical implementations may have to include all production rules in one parser.

The flat form syntax is shown first below as it is a subset of the differential form, for which only the changes are shown after. Tool implementers may handles this syntax variation in different ways, but the easiest route may be that used in the reference parser, which is a single syntax within a parser to which a flag is passed indicating the differential/flat status of the archetype text to be parsed.

```
----------------------- flat form of syntax ----------------------
input:
  c_complex_object
| error

c_complex_object:
  c_complex_object_head SYM_MATCHES SYM_START_CBLOCK c_complex_object_body
SYM_END_CBLOCK

c_complex_object_head:
  c_complex_object_id c_occurrences

c_complex_object_id:
  type_identifier
| type_identifier V_LOCAL_TERM_CODE_REF

c_complex_object_body:
  c_any
| c_attributes

c_object:
  c_complex_object
| archetype_internal_ref
| archetype_slot
| constraint_ref
| c_primitive_object
| V_C_DOMAIN_TYPE
| ERR_C_DOMAIN_TYPE
| error

archetype_internal_ref:
  SYM_USE_NODE type_identifier c_occurrences absolute_path
| SYM_USE_NODE 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 type_identifier
| SYM_ALLOW_ARCHETYPE 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:
  *

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
| error

c_includes:
  -/-
| SYM_INCLUDE assertions

c_excludes:
  -/-
| SYM_EXCLUDE assertions
```

```
c_existence:
  -/-
| SYM_EXISTENCE SYM_MATCHES SYM_START_CBLOCK existence_spec SYM_END_CBLOCK

existence_spec:
  V_INTEGER
| V_INTEGER SYM_ELLIPSIS V_INTEGER

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:
  -/-
| SYM_OCCURRENCES SYM_MATCHES SYM_START_CBLOCK occurrence_spec
SYM_END_CBLOCK
| SYM_OCCURRENCES error

occurrence_spec:
  cardinality_limit_value
| V_INTEGER SYM_ELLIPSIS cardinality_limit_value

c_integer_spec:
  integer_value
| integer_list_value
| integer_interval_value
| occurrence_spec

c_integer:
  c_integer_spec
| c_integer_spec ; integer_value
| c_integer_spec ; error

c_real_spec:
  real_value
| real_list_value
| real_interval_value

c_real:
  c_real_spec
| c_real_spec ; real_value
| c_real_spec ; error

c_date_constraint:
  V_ISO8601_DATE_CONSTRAINT_PATTERN
| date_value
| date_interval_value

c_date:
  c_date_constraint
| c_date_constraint ; date_value
| c_date_constraint ; error

c_time_constraint:
  V_ISO8601_TIME_CONSTRAINT_PATTERN
| time_value
| time_interval_value

c_time:
  c_time_constraint
| c_time_constraint ; time_value
| c_time_constraint ; error

c_date_time_constraint:
  V_ISO8601_DATE_TIME_CONSTRAINT_PATTERN
| date_time_value
| date_time_interval_value

c_date_time:
  c_date_time_constraint
| c_date_time_constraint ; date_time_value
| c_date_time_constraint ; error

c_duration_constraint:
  duration_pattern
| duration_pattern '/' duration_interval_value
| duration_value
| duration_interval_value

duration_pattern:
  V_ISO8601_DURATION_CONSTRAINT_PATTERN

c_duration:
  c_duration_constraint
| c_duration_constraint ; duration_value
| c_duration_constraint ; error

c_string_spec:
  V_STRING
| string_list_value
| string_list_value , SYM_LIST_CONTINUE
| V_REGEXP

c_string:
  c_string_spec
| c_string_spec ; string_value
| c_string_spec ; error

c_boolean_spec:
  SYM_TRUE
| SYM_FALSE
| SYM_TRUE , SYM_FALSE
| SYM_FALSE , SYM_TRUE

c_boolean:
  c_boolean_spec
| c_boolean_spec ; boolean_value
| c_boolean_spec ; error

constraint_ref:
  V_LOCAL_TERM_CODE_REF

any_identifier:
  type_identifier
| V_ATTRIBUTE_IDENTIFIER

-- for string_value etc, see dADL spec

-- for attribute_path, abslute_path, call_path, etc, see Path spec

-- for assertions, assertion, see Assertion spec

----------- rule variations for differential form of syntax -----------
c_complex_object_id:
  type_identifier
| type_identifier V_LOCAL_TERM_CODE_REF
| sibling_order type_identifier V_LOCAL_TERM_CODE_REF

c_archetype_slot_id:
  SYM_ALLOW_ARCHETYPE type_identifier
| sibling_order SYM_ALLOW_ARCHETYPE type_identifier
| SYM_ALLOW_ARCHETYPE type_identifier V_LOCAL_TERM_CODE_REF
| sibling_order SYM_ALLOW_ARCHETYPE type_identifier V_LOCAL_TERM_CODE_REF
| SYM_ALLOW_ARCHETYPE error

sibling_order:
  SYM_AFTER V_LOCAL_TERM_CODE_REF
| SYM_BEFORE V_LOCAL_TERM_CODE_REF
```

B.2.2 Symbols

The following shows the lexical specification for the cADL grammar. The keywords `SYM_BEFORE` and `SYM_AFTER` occur only in the differential form of the syntax.

```
----------/* definitions */ -----------------------------------------------
ALPHANUM [a-zA-Z0-9]
IDCHAR [a-zA-Z0-9_]
NAMECHAR [a-zA-Z0-9._\-]
NAMECHAR_SPACE [a-zA-Z0-9._\- ]
NAMECHAR_PAREN [a-zA-Z0-9._\-()]

UTF8CHAR (([\xC2-\xDF][\x80-\xBF])|(\xE0[\xA0-\xBF][\x80-\xBF])|([\xE1-
\xEF][\x80-\xBF][\x80-\xBF])|(\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])|([\xF1-
\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]))

----------/* 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

"??"        -- -> SYM_DT_UNKNOWN
"?"         -- -> Question_mark_code

"|"         -- -> SYM_INTERVAL_DELIM

"["         -- -> Left_bracket_code
"]"         -- -> Right_bracket_code

"{"         -- -> SYM_START_CBLOCK
"}"         -- -> SYM_END_CBLOCK

".."        -- -> SYM_ELLIPSIS
"..."       -- -> SYM_LIST_CONTINUE
----------/* common keywords */ ---------------------------------------

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

[Ii][Ss]_[Ii][Nn]           -- -> SYM_MATCHES
----------/* assertion 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

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

[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

[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

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

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

[Aa][Ff][Tt][Ee][Rr]                                -- -> SYM_AFTER

[Bb][Ee][Ff][Oo][Rr][Ee]                            -- -> SYM_BEFORE


----------/* V_URI */ ---------------------------------------------
[a-z]+:\/\/[^<>|\\{}^~"\[\] ]*{

---------/* V_QUALIFIED_TERM_CODE_REF */ ----------------------------
-- any qualified code, e.g. [local::at0001], [local::ac0001], [loinc::700-0]
--
\[{NAMECHAR_PAREN}+::{NAMECHAR}+\]
\[{NAMECHAR_PAREN}+::{NAMECHAR_SPACE}+\]         -- error

---------/* V_LOCAL_TERM_CODE_REF */ ---------------------------------
-- any unqualified code, e.g. [at0001], [ac0001], [700-0]
--
\[{ALPHANUM}{NAMECHAR}*\]

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

---------/* V_TERM_CODE_CONSTRAINT of form */ ------------
-- [terminology_id::code, -- comment
--            code, -- comment
--             code] -- comment
--
-- Form with assumed value
-- [terminology_id::code, -- comment
--           code; -- comment
--           code] -- an optional assumed value
--

\[[a-zA-Z0-9()._\-]+::[ \t\n]*        -- start IN_TERM_CONSTRAINT

<IN_TERM_CONSTRAINT> {
    [ \t]*[a-zA-Z0-9._\-]+[ \t]*;[ \t\n]*
       -- match second last line with ';' termination (assumed value)

    [ \t]*[a-zA-Z0-9._\-]+[ \t]*,[ \t\n]*
          -- match any line, with ',' termination

    \-\-[^\n]*\n -- ignore comments

    [ \t]*[a-zA-Z0-9._\-]*[ \t\n]*\] -- match final line, terminating in ']'

------/* V_ISO8601_EXTENDED_DATE_TIME */ ---
-- YYYY-MM-DDThh:mm:ss[,sss][Z|+/-nnnn]
--
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-
9]+)?(Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})? |
[0-9]{4}-[0-1][0-9]-[0-3][0-9]T[0-2][0-9](Z|[+-][0-9]{4})?
----------/* V_ISO8601_EXTENDED_TIME */ --------
-- hh:mm:ss[,sss][Z|+/-nnnn]
--
[0-2][0-9]:[0-6][0-9]:[0-6][0-9](,[0-9]+)?(Z|[+-][0-9]{4})? |
[0-2][0-9]:[0-6][0-9](Z|[+-][0-9]{4})?

----------/* V_ISO8601_DATE YYYY-MM-DD */ --------------------
[0-9]{4}-[0-1][0-9]-[0-3][0-9] |
[0-9]{4}-[0-1][0-9]

----------/* V_ISO8601_DURATION */ -------------------------
P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?T([0-9]+[hH])?([0-9]+[mM])?([0-9]+[sS])? |
P([0-9]+[yY])?([0-9]+[mM])?([0-9]+[wW])?([0-9]+[dD])?

----------/* V_ISO8601_DATE_CONSTRAINT_PATTERN */ -----------------
[yY][yY][yY][yY]-[mM?X][mM?X]-[dD?X][dD?X]

----------/* V_ISO8601_TIME_CONSTRAINT_PATTERN */ ------------------
[hH][hH]:[mM?X][mM?X]:[sS?X][sS?X]

----------/* V_ISO8601_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_DURATION_CONSTRAINT_PATTERN */ --------------
P[yY]?[mM]?[wW]?[dD]?T[hH]?[mM]?[sS]? |
P[yY]?[mM]?[wW]?[dD]?

----------/* V_TYPE_IDENTIFIER */ ------------------------------------
[A-Z]{IDCHAR}*
```

```
----------/* V_GENERIC_TYPE_IDENTIFIER */ ----------------------------
[A-Z]{IDCHAR}*<[a-zA-Z0-9,_<>]+>

----------/* V_FEATURE_CALL_IDENTIFIER */ ----------------------------
[a-z]{IDCHAR}*[ ]*\(\)

----------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------
[a-z]{IDCHAR}*

----------/* V_GENERIC_TYPE_IDENTIFIER */ -------------------------------
[A-Z]{IDCHAR}*<[a-zA-Z0-9,_<>]+>

----------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
[a-z]{IDCHAR}*

----------/* V_C_DOMAIN_TYPE - sections of dADL syntax */ ---------------
-- {mini-parser specification}
-- this is an attempt to match a dADL section inside cADL. It will
-- probably never work 100% properly since there can be '>' inside "||"
-- ranges, and also strings containing any character, e.g. units string
-- contining "{}" chars. The real solution is to use the dADL parser on
-- the buffer from the current point on and be able to fast-forward the
-- cursor to the last character matched by the dADL scanner

-- the following version matches a type name without () and is deprecated
[A-Z]{IDCHAR}*[ \n]*<                 -- match a pattern like
                                      -- 'Type_Identifier whitespace <'
-- the following version is correct ADL 1.4/ADL 1.5
\([A-Z]{IDCHAR}*\)[ \n]*<             -- match a pattern like
                                      -- '(Type_Identifier) whitespace <'

<IN_C_DOMAIN_TYPE> {
    [^}>]*>[ \n]*[^>}A-Z]                   -- match up to next > not
                                            -- followed by a '}' or '>'

    [^}>]*>+[ \n]*[}A-Z]                    -- final section - '...>
                                            -- whitespace } or beginning of
                                            -- a type identifier'

    [^}>]*[ \n]*}                           -- match up to next '}' not
}                                           -- preceded by a '>'

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

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

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

----------/* V_REAL */ -----------------------------------------------
[0-9]+\.[0-9]+
[0-9]+\.[0-9]+[eE][+-]?[0-9]+
----------/* V_STRING */ -----------------------------------------------
\"[^\\\n"]*\"

\"[^\\\n"]*{                 -- beginning of a multi-line string
<IN_STR> {
    \\\\                     -- match escaped backslash, i.e. \\ -> \
    \\\"                     -- match escaped double quote, i.e. \” -> “
    {UTF8CHAR}+              -- match UTF8 chars
    [^\\\n"]+                -- match any other characters
    \\\n[ \t\r]*             -- match LF in line
    [^\\\n"]*\"              -- match final end of string

    .|\n     |
    <<EOF>>                  -- unclosed String -> ERR_STRING
}
```

B.3 Assertion Syntax

The assertion grammar is part of the cADL grammar, which is available as an HTML document. This grammar is implemented and tested using lex (.l file) and yacc (.y file) specifications for in the Eiffel programming environment. The current release of these files is available at http://www.openehr.org/svn/ref_impl_eiffel/TRUNK/components/adl_parser/src/syntax/cadl/parser. The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

B.3.1 Grammar

The following provides the cADL parser production rules (yacc specification) as of revision 166 of the Eiffel reference implementation repository (http://www.openehr.org/svn/ref_impl_eiffel). Note that because of interdependcies with path and assertion production rules, practical implementations may have to include all production rules in one parser.

```
assertions:
  assertion
| assertions assertion

assertion:
  any_identifier : boolean_expression
| boolean_expression
| any_identifier : error

boolean_expression:
  boolean_leaf
| boolean_node

boolean_node:
  SYM_EXISTS absolute_path
| SYM_EXISTS error
| relative_path SYM_MATCHES SYM_START_CBLOCK c_primitive SYM_END_CBLOCK
| SYM_NOT boolean_leaf
| 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_IMPLIES boolean_expression

boolean_leaf:
   ( boolean_expression )
| SYM_TRUE
| SYM_FALSE

arithmetic_expression:
  arithmetic_leaf
| arithmetic_node

arithmetic_node:
  arithmetic_expression + arithmetic_leaf
| arithmetic_expression - arithmetic_leaf
| arithmetic_expression * arithmetic_leaf
| arithmetic_expression / arithmetic_leaf
| arithmetic_expression ^ arithmetic_leaf

arithmetic_leaf:
  ( arithmetic_expression )
| integer_value
| real_value
| absolute_path
```

B.4 Path Syntax

The Path syntax used in ADL is defined by the grammar below. This grammar is implemented using lex (.l file) and yacc (.y file) specifications for in the Eiffel programming environment. The current release of these files is available at http://www.openehr.org/svn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path. The .l and .y files can easily be converted for use in another yacc/lex-based programming environment. The ADL path grammar is available as an HTML document.

B.4.1 Grammar

The following provides the ADL path parser production rules (yacc specification) as of revision 166 of the Eiffel reference implementation repository (http://www.openehr.org/svn/ref_impl_eiffel).

```
input:
  movable_path
| absolute_path
| relative_path
| error

movable_path:
  SYM_MOVABLE_LEADER relative_path

absolute_path:
  / relative_path
| absolute_path / relative_path

relative_path:
  path_segment
| relative_path / path_segment

path_segment:
  V_ATTRIBUTE_IDENTIFIER V_LOCAL_TERM_CODE_REF
| V_ATTRIBUTE_IDENTIFIER
```

B.4.2 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

"//"        SYM_MOVABLE_LEADER

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

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

B.5 ADL Syntax

The following syntax and lexical specification are used to process an entire ADL file. Their main job is reading the header items, and then cutting it up into dADL, cADL and assertion sections. With the advent of ADL2, this will no longer be needed, since every ADL text will in fact just be a dADL text containing embedded sections in cADL and the assertion syntax.

See the *open*EHR Archetype Object Model (AOM) for more details on the ADL parsing process.

B.5.1 Grammar

This section describes the ADL grammar, as implemented and tested in revision 166 of the Eiffel reference implementation repository (http://svn.openehr.org/ref_impl_eiffel).

```
input:
  archetype
| error

archetype:
  arch_identification
  arch_specialisation
  arch_concept
  arch_language
  arch_description
  arch_definition
  arch_invariant
  arch_ontology

arch_identification:
  arch_head V_ARCHETYPE_ID
| SYM_ARCHETYPE error

arch_head:
  SYM_ARCHETYPE
| SYM_ARCHETYPE arch_meta_data

arch_meta_data:
  ( arch_meta_data_items )

arch_meta_data_items:
  arch_meta_data_item
| arch_meta_data_items ; arch_meta_data_item

arch_meta_data_item:
  SYM_ADL_VERSION = V_VERSION_STRING
| SYM_IS_CONTROLLED

arch_specialisation:
  -/-
| SYM_SPECIALIZE V_ARCHETYPE_ID
| SYM_SPECIALIZE error

arch_concept:
  SYM_CONCEPT V_LOCAL_TERM_CODE_REF
| SYM_CONCEPT error

arch_language:
  SYM_LANGUAGE V_DADL_TEXT
```

```
    | SYM_LANGUAGE error

arch_description:
      -/-
    | SYM_DESCRIPTION V_DADL_TEXT
    | SYM_DESCRIPTION error

arch_definition:
      SYM_DEFINITION V_CADL_TEXT
    | SYM_DEFINITION error

arch_invariant:
      -/-
    | SYM_INVARIANT V_ASSERTION_TEXT
    | SYM_INVARIANT error

arch_ontology:
      SYM_ONTOLOGY V_DADL_TEXT
    | SYM_ONTOLOGY error
```

B.5.2 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
    -- mini-parser to match V_DADL_TEXT

^[Ll][Aa][Nn][Gg][Uu][Aa][Gg][Ee][ \t\r]*\n                  SYM_LANGUAGE
    -- mini-parser to match V_DADL_TEXT

^[Dd][Ee][Ss][Cc][Rr][Ii][Pp][Tt][Ii][Oo][Nn][ \t\r]*\n      SYM_DESCRIPTION
    -- mini-parser to match V_CADL_TEXT

^[Ii][Nn][Vv][Aa][Rr][Ii][Aa][Nn][Tt][ \t\r]*\n              SYM_INVARIANT
```

```
        -- mini-parser to match V_ASSERTION_TEXT

    ^[Oo][Nn][Tt][Oo][Ll][Oo][Gg][Yy][ \t\r]*\n                    SYM_ONTOLOGY
        -- mini-parser to match V_DADL_TEXT


    ----------/* V_DADL_TEXT */ -------------------------------------
    <IN_DADL_SECTION>{
        -- the following 2 patterns are a hack, until ADL2 comes into being;
        -- until then, dADL blocks in an archetype finish when they
        -- hit EOF, or else the 'description' or 'definition' keywords.
        -- It's not nice, but it's simple ;-)
        -- For both these patterns, the lexer has to unread what it
        -- has just matched, store the dADL text so far, then get out
        -- of the IN_DADL_SECTION state
        ^[Dd][Ee][Ff][Ii][Nn][Ii][Tt][Ii][Oo][Nn][ \t\r]*\n
        ^[Dd][Ee][Sc][Rr][Ii][Pp][Tt][Ii][Oo][Nn][ \t\r]*\n

        [^\n]+\n                -- any text on line with a LF
        [^\n]+                  -- any text on line with no LF
        <<EOF>>                 -- (escape condition)
        (.|\n)                  -- ignore unmatched chars
    }

    ----------/* V_CADL_TEXT */ -------------------------------------
    <IN_CADL_SECTION>{
        ^[ \t]+[^\n]*\n         -- non-blank lines
        \n+                     -- blank lines
        ^[^ \t]                 -- non-white space at start (escape condition)
    }

    ----------/* V_ASSERTION_TEXT */ -------------------------------------
    <IN_ASSERTION_SECTION>{
        ^[ \t]+[^\n]*\n         -- non-blank lines
        ^[^ \t]                 -- non-white space at start (escape condition)
    }

    ----------/* V_VERSION_STRING */ -------------------------------------
    [0-9]+\.[0-9]+(\.[0-9]+)*                              V_VERSION_STRING

    ----------/* 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_]+){2}\.[a-zA-Z][a-zA-Z0-9_]+(-[a-
    zA-Z][a-zA-Z0-9_]+)*\.v[1-9][0-9]*
                                                          V_ARCHETYPE_ID

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

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 Hein J L. *Discrete Structures, Logic and Computability (2nd Ed)*. Jones and Bartlett 2002.

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

10 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).

11 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.

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

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

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

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

Resources

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

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

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

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

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

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

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

END OF DOCUMENT