# Overview

The following table describes the deliverables of the openEHR specification project and indicates the current status of each. The meanings of the statuses are as follows:

- **stable**: the deliverable has been reviewed and validated by some level of implementation.
- **draft**: the deliverable is available but should not be considered stable yet, as testing of formal semantics is not yet complete.
- **forthcoming**: the deliverable is not available yet, but will be in a future release.
- **under redevelopment**: the deliverable exists but needs to be upgraded or updated.

## Reading the Documents

Most links in the table below are to Adobe PDF files. All files are in colour. If you do not see them in colour or have other problems reading them, we suggest upgrading to the latest Acrobat Reader. If you still experience problems with reading PDF files, your browser configuration may need to be adjusted. See the Adobe Acrobat support page for more help.

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Description</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introducing openEHR</td>
<td>First introduction to the openEHR Foundation and its activities.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td>CM Plan</td>
<td>Technical document describing how versioning, changes, and releases are made.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Requirements</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISO 18308 Conformance Statement</td>
<td>Document describing conformance of openEHR architecture to ISO TS 18308, &quot;Requirements for EHR Architectures&quot;.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Architecture Overview</td>
<td>&quot;Read me 1st&quot; document for the whole architecture. provides a summary of the reference, archetype and service models, and describes global semantics.</td>
<td>Stable</td>
<td>This is a new document containing some content from the old &quot;roadmap&quot; document as well as new content describing the key semantics of the openEHR reference</td>
</tr>
<tr>
<td><strong>Modelling Guide</strong></td>
<td>Describes the use of UML and other related issues in object-oriented modelling.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td><strong>Terminology</strong></td>
<td>Documentary form of the openEHR terminology, which is a set of vocabularies and code sets used by the reference and archetype models.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Reference Model</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EHR IM</strong></td>
<td>The information model of the openEHR EHR</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>EHR Extract IM</strong></td>
<td>The information model of the EHR Extract, which is a serialization of content from an EHR.</td>
<td>Under Redevelopment</td>
<td></td>
</tr>
<tr>
<td><strong>Demographic IM</strong></td>
<td>The openEHR demographic model.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Common IM</strong></td>
<td>Information model containing common concepts, including the archetype-enabling LOCATABLE class, party references, audits and attestations, change control, and authored resources.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Data Structures IM</strong></td>
<td>Information model of data structures, including a powerful model of HISTORY and EVENT.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Data Types IM</strong></td>
<td>Information model of data types, including quantities, date/times, plain and coded text, time specification, multimedia and URIs.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Support IM</strong></td>
<td>Support model containing low-level concepts, assumed types, and terminology interface specification used in the rest of the models.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td><strong>Integration IM</strong></td>
<td>Model supporting expression of legacy data in a free form for further processing into and out of openEHR information structures.</td>
<td>Draft</td>
<td></td>
</tr>
<tr>
<td><strong>Archetype Model</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Archetype Principles</td>
<td>Semantic principles of archetypes and templates.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Archetype Definition Language 1.3 (ADL)</td>
<td>Syntax specification for archetypes.</td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td>Archetype Definition Language 2.0 (ADL2)</td>
<td>Improved syntax specification for archetypes.</td>
<td>Draft</td>
<td></td>
</tr>
</tbody>
</table>

ADL 1.3 remains the current ADL syntax in use, and most current tools only process ADL 1.3. Some improvements that were done as part of the ADL 2 work will be reverse-engineered into ADL 1.3 in the near future; these are mainly to do with pathing.

ADL version 2.0 is an improved syntax for expressing archetypes, and is has two important features over ADL 1.3: it is extensible without the syntax needing to be changed, because in ADL2, an archetype is a dADL document; and due to being in dADL/embedded cADL format, it can more easily be converted to and from XML. The current version of this specification has not yet been fully
<table>
<thead>
<tr>
<th>Specification</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Archetype Object Model (AOM)</strong></td>
<td>Object model of archetypes.</td>
<td>Stable</td>
</tr>
<tr>
<td><strong>Template Object Model (TOM)</strong></td>
<td>Object model of templates.</td>
<td>Forthcoming</td>
</tr>
<tr>
<td><strong>openEHR Archetype Profile (OAP)</strong></td>
<td>openEHR plug-in additions to the generic archetype object model.</td>
<td>Stable</td>
</tr>
<tr>
<td><strong>Archetype System</strong></td>
<td>Description of system of archetype management and governance.</td>
<td>Under Redevelopment</td>
</tr>
<tr>
<td><strong>Service Model</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EHR Kernel API</strong></td>
<td>API of a client component that provides a virtual EHR interface, including fine-grained archetype-based data manipulation, and querying.</td>
<td>Forthcoming</td>
</tr>
<tr>
<td><strong>EHR Service</strong></td>
<td>Coarse-grained service specification for EHR back-end providing check-out, Contribution check-in, querying.</td>
<td>Forthcoming</td>
</tr>
<tr>
<td><strong>Demographic Service</strong></td>
<td>Service specification for demographic back-end.</td>
<td>Forthcoming</td>
</tr>
<tr>
<td><strong>Archetype service</strong></td>
<td>Archetype repository service specification.</td>
<td>Forthcoming</td>
</tr>
<tr>
<td><strong>Computable</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Release Strategy

With the advent of Release 1.0 of openEHR, more rigorous change management rules come into play. These are designed to protect developers and users from adverse effects of changes, and to allow them to upgrade in an orderly fashion. Future releases will follow a 3-digit numbering scheme similar to many open source projects, e.g. Apache, using identifiers like 1.0.1 etc. The meaning of a change in each digit is as follows:

- **3rd position**: used to indicate error corrections and minor additions that do not change the semantics. Thus, Release 1.0.2 is the second error correction release after Release 1.0.
- **2nd position**: used to indicate significant additions that do not change the semantics of the existing part of the release. Release 1.3.0 would be the 3rd release containing compatible additions to Release 1.0.
- **1st position**: used to indicate changes to the semantics or large changes. Release 2.0 would contain changes incompatible in some way with Release 1.0, most likely requiring software upgrade and possibly data migration.

### Changes to Documentation

Where changes to documentation are made, e.g. due to a request to clarify an explanation, fix a typographical error, a CR will be raised, and the revision number of the affected document(s) will change, but there will not be a new release number.

### Error corrections

Where the changes made are to correct an error in a model, parsing rules or some other aspect of the formal semantics of the specifications (and possibly accompanying changes to explanatory text), an error-correction release will be made.

### Compatible Additions

Where the changes have the effect of adding a new specification or other artifact which is completely compatible with the current release, an enhancement release is made.

### Major Changes

Where changes actually alter semantics of existing artefacts (other than for minor error corrections), a new major release is declared. Such changes would normally be grouped into as few such releases as possible.

### Change Management

As with pre-1.0 releases, change management will remain a disciplined process. Two types of online document, the Problem Report (PR) and the Change Request (CR), are used to track problems and
changes. These are used as follows:

- anyone can [raise a PR](#) to indicate some issue or problem with the current release. A PR documents the issue seen from the user's perspective, with the resolution being provided by the development team.
- only a member of the development team can raise a CR. All CRs can be [viewed here](#). A CR documents a change to the current release. A CR will either describe the problem it is solving, or refer to one or more PRs it is designed to address. The CR is a document of the change and the process of applying it to the release.

From Release 1.0 onward, the way CRs are processed will change slightly. All changes to the specifications, apart from document text changes (e.g. improving explanations, fixing typos) will be signalled to the community via the openehr-technical mailing list, and a period of time given for the community to inspect the proposed changes and provide feedback. This is particularly aimed at allowing implementers to indicate the knock-on effects of changes. In the process of such discussions it may turn out that the proposed change will not go ahead, or that it will go ahead in a later release than originally planned. In this way, the community will be able to ensure that releases into the future remain stable and occur at a reasonable rate.

All changes in the Release 1.0 mainline must also be implemented in at least one instance of software, schemas or other appropriate formal expression before being accepted as a change to the specifications.

$LastChangedDate$ $LastChangedRevision$
Introducing openEHR

© 2005 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
T Beale, Dr S Heard, Dr D Kalra, D Lloyd, Dr P Schloeffel

“openEHR” is an internationally registered trademark of the openEHR Foundation

email: info@openEHR.org web: www.openEHR.org
Overview of openEHR

The Foundation

The openEHR Foundation is a not-for-profit company, limited by guarantee. Its founding shareholders are University College London, UK and Ocean Informatics Pty Ltd, Australia. It is regulated under the UK Companies Acts 1985 and 1989. The aims and management of the Foundation, exercised through its Board of Directors, are described on the openEHR website. The Board is responsible for the governance of the Foundation, including strategic direction, financial management, legal regulation, and intellectual property (IP) management. The Board determines the roles, structures, and procedures of the Foundation and ensures that these function correctly. The major work of technical and clinical oversight and supervision of openEHR product developments is delegated to the Architectural Review Board (ARB) and Clinical Review Board (CRB).

Aims

The openEHR Foundation is dedicated to the development of an open, interoperable health computing platform, of which a major component is clinically effective and interoperable electronic health care records (EHRs). It does this by researching clinical requirements, and creating specifications and implementations. The specifications take the form of modular information models, service models and clinical information models. The platform supports the following requirements:

- ability to record any clinical information, including complex time-based lab results, imaging, diagnoses, care plans, evaluations, patient education material, and stateful, workflow-based instructions and intervention information;
- archetype- and template-enabling of all clinical systems, empowering clinical professionals to define the content, semantics and user interfaces of systems independently from the software;
- proper integration with terminology systems, including with: SNOMED-CT\(^1\) so that reliable inferencing and decision support based on EHR data will be possible; LOINC\(^2\), so that traceability and sharing of laboratory data is possible; and ICDx\(^3\) and ICPC\(^4\) classifications, enabling reliable reimbursement, management, and public health studies;
- ability to integrate openEHR with messaging systems, particularly HL7 version 2 and EDIFACT, via the use of “legacy archetypes” and systematic mapping definitions;
- ability to integrate with existing hospital information systems and other databases, also via the use of legacy archetypes;
- integration with applications via a published API;
- distributed versioning and merging of EHR, demographic and other information;
- to make the architecture componentised, adaptive and future-proof, so that it may be a reliable basis for managing 100 year+ health records.

The Foundation publishes the specifications for the architecture openly. It also publishes “implementation technology specification” expressions, such as XML and database schemas corresponding to

---

1. A major ontological terminology effort by the College of American Pathologists; see http://www.snomed.org
the models. From these artifacts it creates open source implementations which are validated in clinical environments, used for further research, and ultimately deployed in actual clinical environments.

The other major activity of the openEHR Foundation is the development of evidence-based clinical information and workflow models, known as archetypes. These are developed collaboratively by domain experts, and published openly in an online, intelligent repository, and are directly consumable by systems based on the openEHR archetype principle.

Modus Operandi
The openEHR Foundation works in an open manner, based on active relationships with domain experts and users, with national and international standards bodies, including ISO, CEN, and sHL7, with software and system developers, and with educational institutions and researchers. The technical work is carried out within a scientific framework i.e. the use of an experimental methodology to validate and refine hypotheses. It is performed in an engineering mode, i.e. with a dedicated team undertaking an iterative process of requirements capture, design, implementation and testing. All of its deliverables are version-controlled and formally change-managed.

Unlike closed commercial developments, openEHR conducts its development in the open, with direct community involvement in specification, software implementation and evaluation, so that users can see and trust the foundations and structures of the software they are using.

Membership
Membership of openEHR implies a commitment towards realising the vision of high quality, interoperable EHRs, and a willingness to share ideas and experience. Membership is free, and is available simply by registration on the openEHR.org website.

openEHR Activities
The openEHR Foundation works in two broad activity areas: the “technical” and the “clinical”. The technical area is where engineering work is done, including specifications, implementations, testing and conformance. The clinical area is where healthcare domain professionals and organisations engage with openEHR, including on the development and deployment of ontologies, archetypes, templates, guidelines, and clinical education and training. These two areas of activity in openEHR are visible in the “two-level modelling” approach of openEHR, through which a rigorous yet flexible development framework enables reliable sharing of clinical meaning in addition to guaranteed data interoperability.

Management
Within each of the two openEHR activity areas, work is performed by project groups (PGs) each comprising a team of developers responsible for the work done on a project. Change management on some of these projects is overseen by a review board. In the technical area, the Architectural Review Board (ARB) performs this function. Its membership comprises some eight members of openEHR, appointed by the Foundation Board, all with long-term experience in an area of health informatics. The current membership of the ARB is posted on the openEHR website ARB page. The role of the ARB is to review and make decisions on change requests (CRs) for reference projects (defined below). It operates using simple majority voting.

In the clinical area, the Clinical Review Board (CRB) performs a corresponding change management function. The CRB, likewise, comprises an international group of experienced clinical professional
members of openEHR. Issues affecting both areas are managed through consultation between the board.

The management structure of openEHR is illustrated in FIGURE 1. Projects whose change is managed by the ARB or CRB are shown joined to the relevant board of review; others are not.

What is an “openEHR Project”?

An openEHR project is defined as any project that:

• is based on openEHR in one of the following ways:
  - contributes to openEHR specifications or
  - aims to build something that satisfies openEHR conformance criteria of nominated openEHR specifications;
• agrees to use the relevant Technical or Clinical Change Management plan (CM Plan) and;
• agrees to the IP framework defined by openEHR for its products and services;
• is registered as an openEHR project with the openEHR Foundation.

Common aspects of openEHR change management in both activity areas include:

• the use of version and change management toolset/environment;
• the use of Problem Reports (PR) and Change Requests (CR), and the lifecycle and process for handling them;
• the use of defined operating procedures including for the distribution and publication of artefacts (also described in the CM plans).

Similarly, openEHR projects agree to respect the following intellectual property rights.

• All project documents must use the openEHR public licence (document form).
• Project source code must use the openEHR open source licence.
• The project must respect and preserve the structure of the org.openehr namespace, which is managed by the ARB.
• Irrevocability: organisations contributing to projects cannot retrospectively revoke the right of the openEHR Foundation and community to continue to use software or other artefacts.

FIGURE 1 Management Structure of openEHR
which they have developed within the openEHR environment (since this would contravene the terms of the license). They may of course use any such developed works as a basis for other developments. This condition ensures that neither the community (which may have come to rely on a component) nor the original developing organisation (which may have spent significant time and money on the development) lose access to the work; if the interests cease to coincide, the development is simply “forked”, and only one line remains within the openEHR framework.

Copyright of donated IP may optionally be transferred to the openEHR Foundation, converted to joint copyright with the openEHR Foundation, or may remain with the originating organisation or author.

Projects agreeing to this framework can take advantage of the facilities provided by the openEHR Foundation, including version management, build servers and a distribution server. It particularly enables smaller projects to proceed where otherwise they might not have sufficient material resources.

This approach to development is offered as a collaboration environment by openEHR, and is not a requirement of developing openEHR-compliant products. Development organisations are welcome to develop openEHR-based products in any way they see fit. It is expected that many projects will be executed by companies, universities and others, according to their own needs and priorities, including fully commercial and closed source projects.

What is an “openEHR Product”?

Regardless of the manner in which its development proceeds, a component, executable, or other artefact (such as an information model schema) can only be represented or promoted as an “openEHR product”, and can only use service marks containing terms such as “openEHR”, “openEHR-compliant”, “openEHR 1.x compliant” and so on, if it has been shown to be conformant to the appropriate openEHR specification(s), through a testing procedure defined and certified by the Foundation, against a specified set of test cases, test data or other appropriate test material. The name “openEHR” is registered internationally as a trademark, and its use with respect to products and services requires permission of the openEHR Foundation. In practical terms this means that users of openEHR-certified end products (particularly those for clinical information management) can rely on claims by the developer to be “openEHR compliant”. It aims to prevent non-conformant (potentially faulty) software or other products being promoted under the banner of openEHR. The test cases and criteria are developed by the community and reviewed by the ARB and/or CRB.

The following figure illustrates the relationship among openEHR projects, products, and non-openEHR work.

![FIGURE 2 Relationship between Projects and Products](image-url)
Intellectual Property

The openEHR Foundation has created a framework to enable the openEHR community to build a repository of intellectual property (IP), for common use. Since the IP includes specifications, software, knowledge bases (e.g. clinical vocabularies) and educational materials, which will find use in clinical and related environments, as well as use in research and education, some legal protection is also required. This is to ensure continuing quality and open access to the IP. The measures adopted by the openEHR Foundation are typical of those used by standards development organisations (due to the fact that openEHR publishes specification documents), and other open source development organisations: copyright, source code licensing, trade- and service-mark protection, and control of the openEHR namespace.

Copyright: Recognition of Authorship

openEHR copyright exists on four kinds of deliverables: documents, software source (including schemas, interfaces), executable software, and knowledge products, such as terminologies. Legally, copyright law guarantees that the original author of the original version of a given artefact is always recognised. However, it does not, on its own, offer much legal protection of such IP, due to the fact that such artefacts keep changing over time, unlike artistic works, which are generally published once only in their finished form (and for which copyright law was mainly developed). For this reason, the openEHR Foundation does not demand that the output of all openEHR projects be copyrighted to the Foundation - the copyright may be retained by the original developer, or a joint copyright may be adopted. However, all reference specifications, ITSs and reference implementation project deliverables must be solely or jointly copyrighted to the openEHR Foundation. Deliverables of non-reference implementation projects may retain the copyright of the original developing organisation.

Source Licenses: Protection for Developers

Because specification documents, software and related artefacts are by their nature constantly changing, the conditions of use, copying, modification and sale are specified explicitly in licenses, rather than relying on potentially unreliable copyright law (which in any case varies across jurisdictions). Two types of license are used - one for documents, and one for software and related materials. The openEHR licenses are available from the Foundation website license page. These licenses are designed to guarantee fair, open and continued availability of all openEHR products to the openEHR community. They provide the main protection for developers of materials that their work will not be taken out of circulation, or otherwise appropriated by private individuals or bodies.

Trademark & Service-marks: Protection for Users

Controlled use of the “openEHR” trademark and related service-marks is the main protection for users of openEHR products, ensuring that, for any product claiming to be compliant, conformant or otherwise based on openEHR, this is in fact the case, and that the exact meaning of the claimed conformance/compliance can be investigated (e.g. by accessing and inspecting the relevant test cases or other materials from the openEHR website). The “openEHR” trademark and any related service-mark may only be used with permission of the openEHR Foundation.

End-use Licenses: Rights of Users

End-use licenses govern the conditions of use of end products - artefacts that can be deployed in a runtime environment. The openEHR Foundation itself employs only a minimal end-use license for its reference implementations, which essentially says that the relevant artefact can not be modified while being presented as its original. This simply ensures that a published component is not altered at the binary level and passed off as the original (or at least that where this happens, it is clearly in violation
of the conditions of use). More complex end use licenses (often called “EULAs” - end use license agreements) may be used by commercial organisations to control the use of their products; such licenses are not the business of the openEHR Foundation.

The openEHR.org Domain and org.openehr Namespace

The openEHR Foundation owns the openEHR.org internet domain, all related variants and sub-domains, and by extension, the org.openehr namespace. Population of the namespace (for example with Java libraries and XML-schemas) is managed on behalf of the Foundation by the openEHR ARB, for the benefit of the community. The principal aim is to ensure that the namespace is clearly defined. Only openEHR reference deliverables may be defined in the org.openehr namespace.

Technical Activities

FIGURE 3 illustrates the areas technical activity of openEHR, including specification and implementation projects, and delivery/deployment activities.

Requirements Project

openEHR undertakes the collation of requirements for the design, implementation and deployment of interoperable EHRs a) to support the seamless sharing and continuity of health care and b) to enable EHR systems to interface with medical knowledge, evidence of best practice and other systems needed to deliver safe, secure and effective health services. Requirements developed by openEHR are contributed to relevant international standards initiatives.

Architecture Project

The openEHR Foundation publishes a number of formal model specifications, including: the openEHR Reference Model (RM), consisting of the primary information models (IMs), the archetype
model (AM) which includes the Archetype Definition Language (ADL) and Archetype Object Model (AOM), and the openEHR service model (SM), which defines interfaces to major software services in a health information environment.

Because these specifications are underpinned by explicit requirements and by the results of implementation and deployment of previous versions and thus constitute an evidence-based information architecture. As with requirements, openEHR architectural specifications are also contributed to relevant international standards initiatives.

Abstract information models are published as directly usable implementation technologies specifications (ITs), such as OMG IDL, XML, programming languages, and database schemas.

Implementation Projects

The openEHR Foundation engages in the implementation of interoperable software components, including archetype and EHR tools and components, using rigorous design and development methodologies. These reference implementations enable the validation of the published specifications, ensuring they are not simply 'paper' exercises.

Standards Activities

The openEHR Foundation is committed to supporting relevant government-sponsored and industry-based standards bodies as a means of encouraging the widespread and effective adoption of interoperable EHRs. Members of openEHR work closely with standardisation bodies, including ISO TC215, CEN TC/251, HL7 and national standards bodies.

Educational Activities

In order to promote and assist understanding and acceptance of openEHR methodology and models by a wide audience, the openEHR Foundation develops educational materials, runs workshops, and provides consultancy services internationally. It also develops materials suitable for use in clinical, healthcare, and health informatics courses.

Clinical Activities

Activities in the openEHR clinical activity area are illustrated in FIGURE 4.
Vocabularies and Terminology

In the clinical activity area, openEHR is concerned with existing and emerging clinical ontologies and terminologies. It also develops specific openEHR terminologies of its own to facilitate the integration of software components and knowledge bases, through the use of archetypes and templates.

Archetypes and Templates

Archetypes are a key element of the openEHR methodology. They are reusable, structured models of clinical information concepts that appear in EHRs, such as 'test result', 'physical examination' and 'medication order', and are expressed in terms of constraints on the reference model. All data in openEHR EHRs are instances of reference model entities, configured by archetypes. Archetypes also act as mediators between data and terminology. They are language- and terminology-neutral.

Templates are (usually) locally defined models of screen forms, and ring together a selection of archetypes, terminologies, language and other details relevant to the particular local use of archetypes. For example, concepts such as 'referral' and 'prescription' are modelled as templates, which in turn use archetypes for more fine-grained concepts.

Educational Activities

The openEHR Foundation develops methodologies and publishes methods and materials for the formulation and use of archetypes, templates, terminologies and clinical guidelines, including those developed by openEHR projects. In collaboration with academic colleagues, it also develops materials suitable for use in clinical, healthcare and post-graduate health informatics courses.

Standards Activities

The openEHR Foundation is working with national and international standards bodies and professional organisations to establish standards for the representation, capture and sharing of clinical knowledge for use in health information environments.

The openEHR Methodology

The openEHR development methodology, designed to formally integrate technical and clinical work in a coherent manner is summarised by FIGURE 5. In the technical environment, modellers and software developers create small, generic models and specifications which are then implemented within systems. They also build tools to support users performing modelling the healthcare domain - the building of archetypes above terminologies and ontologies. At runtime, systems are driven by the definitions created by domain experts using the tools. The information processing capabilities of such systems can evolve smoothly, based mainly on ongoing tool-enabled development work of clinical experts, rather than by continual software maintenance and redeployment.

The key benefits of this approach are:

- direct, flexible and sustainable involvement of domain experts and users;
- significant reductions available in costs of development and deployment of health information systems, due to smaller, more stable software;
- the creation of a much more adaptable and durable health computing environment.
Participating in openEHR

A useful starting point for prospective participants is the openEHR primer. Membership is free, and obtained via the membership page. The Foundation maintains four discussion lists:

- openehr-announce: one way list for major announcements;
- openehr-technical: dedicated to openEHR technical architecture, archetype language, service models and so on; recommended for all ICT professionals working with openEHR, and healthcare professionals interested in technical issues;
- openehr-clinical: dedicated to clinical modelling, archetype development, terminologies, clinical use and demonstration of openEHR; recommended for health professionals and ICT people interested in the clinical world;
- openehr-implementers: dedicated to implementation issues. Recommended for actual and prospective implementers.

Developers are encouraged to visit the developer page. An equivalent starting point for health professionals is the clinicians’ page.

Organisations interested in supporting or contributing to openEHR should contact Dr Dipak Kalra at d.kalra@chime.ucl.ac.uk or info@openEHR.org.
# Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Changes to Aims and Modus Operandi.</td>
<td>T Beale</td>
<td>02 Jan 2006</td>
</tr>
<tr>
<td>1.0</td>
<td>Final group review.</td>
<td>D Ingram, D Kalra, D Lloyd</td>
<td>15 Mar 2005</td>
</tr>
<tr>
<td>0.9.5</td>
<td>David Ingram review.</td>
<td>D Ingram</td>
<td>12 Mar 2005</td>
</tr>
<tr>
<td>0.9.1</td>
<td>Various comments from Dipak.</td>
<td>D Kalra</td>
<td>03 Mar 2005</td>
</tr>
<tr>
<td>0.9</td>
<td>Initial writing. Based on CM Plan and openEHR planning meetings.</td>
<td>T Beale S Heard D Ingram D Kalra D Lloyd</td>
<td>12 Feb 2005</td>
</tr>
</tbody>
</table>
The openEHR Change Management Plan

Editor: {T Beale}¹

Revision: 1.0

Pages: 37

---

1. Ocean Informatics Australia

© 2003-2006 The openEHR Foundation

The openEHR foundation
is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

**Founding Chairman**
David Ingram, Professor of Health Informatics, CHIME, University College London

**Founding Members**
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2005

All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form:
   "© Copyright openEHR Foundation 2001-2005. All rights reserved. www.openEHR.org"

6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
# Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Update release details.</td>
<td>T Beale</td>
<td>28 Jan 2006</td>
</tr>
<tr>
<td>0.9</td>
<td>Changes due to migration to subversion.</td>
<td>T Beale</td>
<td>28 Jul 2005</td>
</tr>
<tr>
<td>0.8</td>
<td>Split off openEHR Overview document.</td>
<td>T Beale</td>
<td>04 Feb 2005</td>
</tr>
<tr>
<td>0.7</td>
<td>Addition of sections on governance, IP and project overview.</td>
<td>T Beale, M Darlison</td>
<td>10 Jan 2005</td>
</tr>
<tr>
<td>0.6</td>
<td>Review by UCL team</td>
<td>D Kalra, T Austin, N Lea, D Lloyd T Beale</td>
<td>10 Dec 2004</td>
</tr>
<tr>
<td>0.5.2</td>
<td>Review by prof. David Ingram (UCL).</td>
<td>D Ingram</td>
<td>20 Feb 2004</td>
</tr>
<tr>
<td>0.5.1</td>
<td>Further review by Tim Cook <a href="mailto:Tim@openparadigms.com">Tim@openparadigms.com</a>.</td>
<td>T Beale</td>
<td>10 Dec 2003</td>
</tr>
<tr>
<td>0.5</td>
<td>Further development of ARB process.</td>
<td>T Beale</td>
<td>06 Dec 2003</td>
</tr>
<tr>
<td>0.4</td>
<td>Review by Tim Cook <a href="mailto:Tim@openparadigms.com">Tim@openparadigms.com</a>; further development.</td>
<td>T Cook, T Beale</td>
<td>22 Nov 2003</td>
</tr>
<tr>
<td>0.3</td>
<td>Updated in preparation for public openEHR CM site launch.</td>
<td>T Beale</td>
<td>15 Nov 2003</td>
</tr>
<tr>
<td>0.2</td>
<td>Updated May visit UCL</td>
<td>T Beale</td>
<td>13 May 2003</td>
</tr>
<tr>
<td>0.1</td>
<td>Initial Writing</td>
<td>T Beale</td>
<td>20 Jan 2003</td>
</tr>
</tbody>
</table>

## Acknowledgements

This paper has been funded by The University College, London and Ocean Informatics, Australia.
1 Introduction

1.1 Purpose

1.2 Audience

1.3 Status

1.4 Terms and Acronyms

2 Overview

2.1 Activities

2.2 Management

3 openEHR Technical Projects

3.1 The Specification project

3.2 Reference Implementation Projects

3.3 Ad hoc Implementation Projects

4 Repository Organisation

4.1 Repository Naming

4.2 Repository Design

5 Release Management

5.1 Overview

5.2 Release Structure and Branching

5.3 Release Naming

6 Change Management

6.1 Overview

6.2 The Change Process

6.2.1 Overview

6.2.2 Problem Reporting

6.2.3 Change Request Process for openEHR Reference Projects

6.2.3.1 Workflow

6.2.3.2 CR Lifecycle

6.2.3.3 Project Group-managed CRs

6.2.3.4 Review Board-managed CRs

6.2.4 Change Request Process for ad hoc Projects

6.2.4.1 Workflow

6.2.4.2 CR Lifecycle

6.2.4.3 CR Management

7 Tools

7.1 Overview

7.2 Configuration Management System

7.3 PR / CR Database

7.4 Publishing/Distribution

8 CI Identification

Appendix A Forms

A.1 Problem Report Form

A.2 PR Form Fields

A.3 Change Request Form

A.4 CR Form Fields
1 Introduction

1.1 Purpose
The purpose of this document is to describe the management of openEHR technical projects, i.e. projects in the openEHR technical space (as described in the document Introducing openEHR) and carried out within the openEHR development environment. These projects have “controlled” deliverables, and a clear problem reporting and change request strategy, defined by this document. Two change management strategies are described: one with a “review board” for reference projects, and one without, for ad hoc projects.

1.2 Audience
The primary audience for this document is developers of specifications and software for the openEHR Foundation.

1.3 Status

1.4 Terms and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARB</td>
<td>Architectural Review Board</td>
</tr>
<tr>
<td>CI</td>
<td>Configuration Item - any controlled artifact, such as a document, source file, test case etc.</td>
</tr>
<tr>
<td>CM</td>
<td>Configuration management</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual property</td>
</tr>
<tr>
<td>PG</td>
<td>Project Group</td>
</tr>
</tbody>
</table>
2 Overview

2.1 Activities

The figure below illustrates the technical activity areas of openEHR, including specification and implementation projects, and delivery/deployment activities.

![Diagram of openEHR Technical Activities]

Each solid-line bubble on the left part of the diagram is a project in openEHR. As shown, there are four major areas of activity: specification, implementation, knowledge, and delivery. The first two correspond to what most people would think of as software development; their change management is the subject of this document. The projects in these two groups are described in more detail in section 3 on page 12.

2.2 Management

In the technical space of openEHR, work is performed by project groups (PGs), which are in some cases overseen by the Architectural Review Board (ARB). The ARB consists of a eight or more international members of openEHR, all with long-term experience in an area of health informatics. The current makeup of the ARB may be found on the openEHR website ARB page. The ARB’s function is to review and make decisions on requests for change that either have significant impact on a project, or that cannot be resolved by the project development group on its own. It operates using simple majority voting.

**Project types**

Project groups are groups of developers responsible for the work done on a project. There are two kinds of openEHR technical project - reference projects, and ad hoc projects. Reference projects develop reference deliverables, which constitute the “official” basis for the community to develop and test the conformance of products. All reference projects are change managed by their project group and the ARB.
Ad hoc openEHR projects on the other hand do not generate reference specifications or implementations, and can be change-managed without recourse to the ARB. Ad hoc project groups will most likely be self-selecting. Anyone can become a member of a project by making themselves known to the existing group, and being given modification rights on the relevant repository. A management view of openEHR’s technical space is illustrated in FIGURE 2 below. In this figure, reference project groups are shown connected to a board of review, while ad hoc projects are not.

**FIGURE 2** Management View of the openEHR Technical Space

**openEHR Reference Deliverables**

Some deliverables created in the openEHR technical space are “reference deliverables”. These artifacts are the definitive instance in their category: the openEHR information and service model specifications, the implementation technology specifications (ITBs) such as XML-schemas, programming language interfaces, openEHR terminology, conformance test cases and openEHR reference implementations (e.g. parsers). Reference implementations are created either for the purposes of conformance testing, or in cases where absolutely dependable, re-usable, standard components are required. All openEHR reference deliverables are created by openEHR reference projects.

The relationship among various kinds of projects and openEHR/non-openEHR products is shown in FIGURE 3 below.

**FIGURE 3** Relationship between Projects and Products

**Definition of an openEHR Technical Project**

For the purposes of managing work done under the openEHR banner, the notion of an “openEHR technical project” is explicitly defined as any project that follows this change management plan.
is a) based on openEHR in some formal way (typically aims to build something that satisfies openEHR conformance criteria) and b) that agrees to do its work within the development framework offered at openEHR.org, defined below. All projects that develop reference deliverables are openEHR projects. The openEHR project development environment is defined by the following.

Change Management

- A standard version and change management toolset/environment. openEHR currently uses the open source tool Subversion for this purpose.
- A basic change management rule - only a member of a project team can a) create a change request, and b) make any change to the project repository. This simply means that the team always knows who is in it, and has agreed among themselves that they can make modifications. Non-team members proposing sensible modifications are likely to be asked to join the team.
- A standard Problem Report (PR) lifecycle.
- A standard Change Request (CR) lifecycle.
- A standard online tool/environment for creating and accessing CRs and PRs. CRs need to be able to be created and viewed in a standard way by developers, whilst PRs need to be created and viewed in a known place and in a sensible way by users. PRs are the public problem-logging and reporting interface for users.
- For reference projects, development is overseen by the ARB, according to the change management process described in Change Request Process for openEHR Reference Projects on page 23. For ad hoc projects, development may adopt the simpler non-ARB process described in Change Request Process for ad hoc Projects on page 26.

Build and Release

- The top level directory structure of implementation projects is fairly similar if not the same.
- An approach to build management that is as far as possible homogeneous across projects (facilitates developers working on more than one project). This does not necessarily have to be a single tool, but if e.g. ant and make are used, they should be used in the same way across projects.
- A standard way of distributing the software to users, particularly binaries (i.e. make it easy for non-IT users).

Intellectual Property Rights

- Copyright may optionally be transferred to the openEHR Foundation, converted to joint copyright with the openEHR Foundation, or may remain with the originating organisation or author.
- Documents (e.g. manuals) use the standard openEHR document licence.
- Source code uses the standard openEHR open source licence. This is currently the Mozilla tri-license, which is really just a meta-license allowing the user to nominate GPL, LGPL, or MPL as the licence they use the software under.
- Respect the structure of the org.openehr namespace by the ARB, for source code, schemas, terminologies and any other reference deliverable. Non-reference projects may not use the org.openehr namespace.
- Irrevocability: organisations cannot retrospectively revoke the right of the openEHR Foundation and community to continue to use software or other artifacts which they have developed within the openEHR environment (since this would contravene the terms of the
license). They may of course use any such developed works as a basis for other developments. This condition ensures that neither the community (which may have come to rely on a component) nor the original developing organisation (which may have spent significant time and money on the development) lose access to the work; if the interests cease to coincide, the development is simply “forked”, and only one line remains with openEHR.

Projects that agree to these items will be able to take advantage of the facilities provided by the openEHR Foundation, including version management, build servers and a distribution server. It particularly enables smaller projects to proceed where otherwise they might not have sufficient technical resources.

This approach to development is offered as a service by openEHR, and of course is not a requirement of developing openEHR-compliant products. Development organisations are encouraged to develop openEHR-based products in any way they see fit. Many projects will be done by companies, universities and so on, according to their own processes, including completely commercial closed source projects.
## 3 openEHR Technical Projects

### 3.1 The Specification project

The openEHR specification project includes deliverables that are considered technical specifications - i.e. that can be used either to develop further specifications, or to build systems, test plans, or other usable artifacts. The specification project includes requirements, abstract architecture, ITSs and conformance specifications as shown in the following table.

<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Requirements Base</td>
<td>The Requirements Base is a repository of requirements underpinning the EHR and related functionality in the health information environment. This repository is the definitive requirements basis of openEHR, and will continue to evolve in time. It consists of both functional requirements and use cases.</td>
</tr>
<tr>
<td>Conformance Statement</td>
<td>Conformance Statement</td>
<td>Conformance statement of openEHR with respect to existing/emerging standards, e.g. ISO TS 18308.</td>
</tr>
<tr>
<td>Architecture</td>
<td>Design Principles</td>
<td>Design principles of health information systems and in particular the EHR</td>
</tr>
</tbody>
</table>
| Service Model (SM)   | Reference Model (RM)    | The primary set of abstract, formal specifications of openEHR models in the information viewpoint. These abstract expressions are independent of implementation technologies. Includes abstract Information Models for:  
  - EHR_extract  
  - Common  
  - Data Structures  
  - Data Types  
  - Support (low level primitives) |
| Archetype Model (AM) | Service Model (SM)      | The primary set of abstract, formal specifications of openEHR models in the computational viewpoint. These abstract expressions are independent of implementation technologies. Includes abstract Service Models for:  
  - EHR  
  - Demographics  
  - Workflow  
  - Archetype repository |
| Archetype System     | Archetype Model (AM)    | Various formal specifications defining the openEHR archetype semantics.  
  - Archetype Principles  
  - Archetype Definition Language (ADL) specification  
  - Template Definition Language (TDL) specification  
  - Archetype Query Language (AQL) specification  
  - Archetype Object Model (AOM) |
|                      | Archetype System        | The openEHR Archetype System                                                                                                               |
The primary models are used as the source for the published documentary form of the specifications, generally in Adobe PDF format. There is not considered to be any semantic difference between tool-based abstract model expressions and their documentary counterparts, i.e. there is no “mapping” or “conversion”. The primary models are also losslessly translated to a UML-2.0 compliant XML instance form, from which all other views are generated.

**Implementation Technology Specification (ITS) Components**

ITSs are the concrete expressions of abstract specifications in specific implementation-oriented technologies, and are the artifacts used directly for building software and databases. They are generated via a mapping process, and may have reduced semantic content, e.g. they might not include certain abstract semantics such as functions, invariants. For example, the XML-schema ITS does not contain functions, since XML-schema is a data-oriented formalism, and does not have a way (or need) to express functions. Otherwise however, ITSs do include full coverage of all the relevant openEHR specifications. The two categories of ITS are as follows.

**Interoperability specifications** include any expression of an abstract specification in a concrete interoperability technology, including:

- IDL expressions, e.g. in OMG IDL syntax, DCE syntax Microsoft, WSDL, or other publicly available interface formalisms
- XML schema or other XML-based formalism

**Implementation specifications** include any expression of an abstract specification in a concrete implementation technology, including:

- any programming language. Such expressions may be code interfaces, example working code, or other code guidelines.
- any database schema language. Such expressions may include full schemas for particular database products and generic schemas for a class of product.

**Change Management**

In the above table, the items in the “Component” column are the items against which problem reports (PRs) are made. In general, PRs will only be raised against executable components or computable components like XML-schemas.

The abstract architecture deliverable (second major row of table) is the main driver of major releases of the specification project. That is to say, if the abstract models change in any significant way, a new release is declared. The Architectural Review Board (ARB) decides on new releases.
Changes in the abstract architecture models will immediately cause changes in the ITSs, since these are the directly usable expression of the abstract architecture. However it might be some months before various implementation projects are changed to reflect specification changes.

### 3.2 Reference Implementation Projects

Reference implementation projects produce artifacts of which only one official version is needed. These are usually used as the basis for conformance testing, or in some cases, are core software components that must have guaranteed correctness and reliability. The following table shows a number of openEHR reference implementation projects.

<table>
<thead>
<tr>
<th>Project</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>(various)</td>
<td>This project develops various tools</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• any converter that creates a derived artifact from an abstract one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• tools for validating specifications in the reference model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• tools for working with test data or test cases</td>
</tr>
<tr>
<td></td>
<td>Kernel in lang X</td>
<td>Open source implementation of the openEHR reference and archetype models, as an archetype- and template-enabled data processing component.</td>
</tr>
<tr>
<td></td>
<td>Wrappers in other languages</td>
<td>Open source implementations of wrappers of the kernel for other development languages</td>
</tr>
<tr>
<td>ADL Parser</td>
<td>Eiffel ADL Parser</td>
<td>Original ADL reference parser implementing current version of the ADL specification.</td>
</tr>
<tr>
<td></td>
<td>Java-wrapped ADL parser</td>
<td>Java wrapping for the parser, implemented using JNI</td>
</tr>
<tr>
<td></td>
<td>dotNet ADL parser</td>
<td>Dotnet edition of the parser, for MS Windows.</td>
</tr>
</tbody>
</table>

### 3.3 Ad hoc Implementation Projects

The *ad hoc* implementation projects produce non-reference tools or systems based on any reference deliverable, whether directly from specifications, or on existing components. Over time, there may well be multiple projects each implementing the same category of deliverable, such as an archetype editor or EHR server; products developed in this way will usually perform the same general function, but may have significantly different performance characteristics, user interface design approaches, or differ in some other way relevant to end use. The following table shows a list of possible openEHR projects, and their components.

<table>
<thead>
<tr>
<th>Project</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archetype Editor</td>
<td></td>
<td>Tool for creating and viewing archetypes.</td>
</tr>
<tr>
<td>EHR server</td>
<td></td>
<td>EHR repository providing versioned Contribution interface, with transaction management</td>
</tr>
<tr>
<td>Demographic server</td>
<td></td>
<td>Demographic data repository, providing versioned Contribution interface, with transaction management</td>
</tr>
<tr>
<td>Project</td>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>etc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4 Repository Organisation

4.1 Repository Naming

Each *openEHR* project is controlled in a separate versioned repository. Currently, the Subversion tool is used to manage repositories (see [http://subversion.tigris.org](http://subversion.tigris.org) and the relevant *openEHR* website pages). Due to the way Subversion works, each project is a separate repository (explained in detail in next section). Further, since most implementation efforts are oriented to one or other of the major technologies available, repositories are also distinguished on this basis. Technologies include:

- Java and related frameworks, such as Eclipse, NetBeans, EJBx, J2EE, Hibernate etc;
- Microsoft .Net technologies, including C#.Net and VB.Net languages;
- Python and related tools such as Zope and Plone;
- The Eiffel language and technologies.

Repositories are separated on the basis of coherent projects rather than fine-grained choices of e.g. persistence framework for Java. Thus, the “Java reference kernel” project might develop one set of base libraries, and two alternative persistence mechanisms. If all of this work forms a coherent whole, it will be defined as a project, and have its own Subversion repository. On the other hand, not all things done in a given technology need be in the same repository. Knowledge tools such as archetype editors and repository tools built in Java would have their own repository, since the project is independent of the EHR kernel and server work, and developed by a different group. There is no problem for sharing libraries from one project with another - this is simply done by setting up the user workspace on the client machine appropriately, so that build scripts can find all the software.

In general the repositories are also designed as “source-only” repositories; there is a separate repository for distributable binaries.

4.2 Repository Design

The *openEHR* subversion repositories include the following:

- **specification**: the specification project
- **knowledge**: clinical domain content, including terminology, archetypes, templates etc
- **knowledge_tools_java**: knowledge management tools built in Java, including a Java archetype editor
- **knowledge_tools_dotnet**: knowledge management tools built in .Net languages, including the Ocean Informatics VB.Net archetype editor
- **ref_impl_java**: Java implementation of *openEHR* reference and archetype models, and archetype-processing kernel
- **ref_impl_dotnet**: Microsoft .Net implementation of *openEHR* reference and archetype models, and archetype-processing kernel
- **ref_impl_eiffel**: reference implementation of *openEHR* in the Eiffel language
- **oe_distrib**: repository containing binaries and packages for distribution.

Other repositories will be defined as the need arises. The full list of projects can always be found on the “projects” button on the *openEHR* home page.

The internal structure of repositories tries to achieve two (sometimes contradictory) aims:
The openEHR Change Management Plan

Repository Organisation
Rev 1.0

- enable basic functioning of Subversion;
- be consistent across repositories, in order to facilitate comprehension by new project members;
- be consistent with accepted conventions for the relevant implementation technology, particularly as required by build tools or other framework components.

In support of the first aim, the following top-level structure is usually used:

- **TRUNK**: mainline current development;
- **BRANCHES**: branch developments for new or alternative or test work;
- **TAGS**: named baselines (no development allowed)
- **RELEASES**: release branches, i.e. branches corresponding to stable major points in the mainline (trunk) development, whose only further allowed changes are bugfixes.

This corresponds more or less to the recommended way in Subversion of separating mainline development; the only difference is that in openEHR upper-case directory names have been chosen to make the special status of these directories clear in a check-out structure, and the RELEASES directory has been added.

Further directory structure is built under the TRUNK directory. The following top-level division may be used:

- **apps**: subdirectories contain sources corresponding to various standalone applications;
- **components**: subdirectories contain sources corresponding to re-usable components, usually in the form of libraries such as dlls etc;
- **libraries**: subdirectories contain sources of libraries which are (re-)used at compile time by other libraries, components or applications;
- **distrib**: sources or files required for building or configuring distributable artefacts.

Below this level, directories would normally have the name of the particular library, component or application. Below that, the following standard names can be used where sensible:

- **src**: source;
- **doc**: documentation of source code;
- **lib**: sources which correspond to a binary library, i.e. a .so or .dll;
- **conf or etc**: configuration files;
- **app**: source corresponding to a standalone application or utility.

The above structures should be treated as guidelines; recommended directory structures are usually available from the documentation of tools being used.
5 Release Management

5.1 Overview

A named “release” is formally defined as a named list - known as a “baseline” - of the set of controlled items in a repository, and their individual version numbers at the point of time of the release. Releases correspond to the release of major additions in specification or functionality of the total product, and usually occur in coarse-grained time, e.g. every quarter, six months, or year. Every change request that is processed between releases is targeted to a particular release, usually the next one, but not necessarily - the CM system allows multiple future releases to be running at once.

Releases are created in Subversion repositories simply by performing an svn copy operation, which essentially creates a “lazy” copy of the current state of a repository. Inspecting the RELEASERS directory in any repository will which release branches have been created; inspecting the TAGS directory will show exactly the release points that have been tagged. For a given release branch, e.g. “release 0.9” of a repository, there may be more than one tag, e.g. “0.9”, “0.9.1”, “0.9b” and so on.

Each release proceeds through a number of phases. The rules about what kind of changes can be made to the repository during the phases vary, as shown in the following example:

- phase = development any change
- phase = test only changes to correct errors or bugs
- phase = production only changes to correct bugs found in use

The actual phases used in openEHR repositories may vary with the repository. In fact, only the difference between “development” and “after development” phases are made by branching the repositories.

All releases are named “release-XXX”, e.g. “release-1.5”, where “1.5” is the release identifier. Release identifiers can be any string.

5.2 Release Structure and Branching

The relationship between releases is worth explaining in some detail, since it is the basis of the workflow of any project. FIGURE 4 illustrates a typical workflow. Typical activities are as follows.

- Work starts on the mainline of a repository (in the TRUNK directory), and continues for some time.
- At some point, it is decided that the state of work is stable enough to declare as a named release that could be used outside the development team. What happens at this point is that a logical “branch” is created, representing the named release, while the mainline evolves as usual. With Subversion, a branch is effected simply by an svn copy of the current TRUNK contents to the a named directory in the RELEASERS directory.
- The directory hierarchy in RELEASERS corresponding to the logical branch is now considered to be limited to that release, i.e. it is considered to be in a testing phase. The only allowable changes to it are bug and documentation fixes.
- Tags can be made of any of the states of a release by performing the appropriate svn copy command from (for example) RELEASERS/release-xyz to TAGS/release-xyz.n.
- Meanwhile, general development continues on the repository mainline in the TRUNK. As time goes on, fixes will accumulate in each release branch made to date, due to testing and use. Usually these fixes will be needed on the mainline development as well; the way to obtain them is to create a “patch” containing changes since the beginning of the branch repository, and apply it to the repository mainline (red dashed arrows right to left).
This operation may be repeated, where each patch is generated from the change following the last change in the previous patch.

Eventually another release will be declared, and the whole operation will repeat, leading to (for example), xxx-dev, xxx-0.9 and xxx-1.0 repositories. As time goes on, users will start using the xxx-1.0 release, and xxx-0.9 will fall into disuse, and could ultimately be declared obsolete (no longer supported) and be archived.

FIGURE 5 illustrates this release logic applied to the openEHR specification repository. The initial repository is spec-dev, i.e. the main line of development in which all kinds of changes are added. At some point it will be cloned into spec-0.9, a branch for the 0.9 release of the openEHR specifications. The only changes permitted to be done to the spec-0.9 repository are those that fix bugs or problems designated to be fixed in release 0.9. At a later point in time, a spec-1.0 repository is created.

The changes made to release-0.9 and release-1.0 may be transmitted one at a time back to specification/TRUNK, by systematic patching, or cumulative patches may be made. Patches may also be made from specification/RELEASES/release-0.9 to specification/RELEASES/release-1.0.
5.3 Release Naming

With the advent of Release 1.0 of openEHR, more rigorous change management rules come into play. These are designed to protect developers and users from adverse effects of changes, and to allow them to upgrade in an orderly fashion. Future releases will follow a 3-digit numbering scheme similar to many open source projects, e.g. Apache, using identifiers like 1.0.1 etc. The meaning of a change in each digit is as follows:

- 3rd position: used to indicate error corrections and minor additions that do not change the semantics. Thus, Release 1.0.2 is the second error correction release after Release 1.0.
- 2nd position: used to indicate significant additions that do not change the semantics of the existing part of the release. Release 1.3.0 would be the 3rd release containing compatible additions to Release 1.0.
- 1st position: used to indicate changes to the semantics or large changes. Release 2.0 would contain changes incompatible in some way with Release 1.0, most likely requiring software upgrade and possibly data migration.

Changes to Documentation

Where changes to documentation are made, e.g. due to a request to clarify an explanation, fix a typographical error, a CR will be raised, and the revision number of the affected document(s) will change, but there will not be a new release number.

Error Corrections

Where the changes made are to correct an error in a model, parsing rules or some other aspect of the formal semantics of the specifications (and possibly also change explanatory text), an error-correction release will be made.

Compatible Additions

Where the changes have the effect of adding a new specification or other artifact which is completely compatible with the current release, an enhancement release is made.

Major Changes

Where changes actually alter semantics of existing artefacts, a new major release is declared. Such changes would normally be grouped into as few such releases as possible.
6  Change Management

6.1  Overview

The approach to change management described here has been developed from change management plans used in a number of industrial contexts. Useful published resources for interested readers include the IEEE standards for configuration management, change management and related issues. A good online resource explaining the concepts at the Technical University of Eindhoven (software engineering home page, CM top page).

FIGURE 6 illustrates the overall openEHR change environment. For each project, a repository that is controlled by the configuration management (CM) system, and whose items (documents, source, etc) are created and modified by project groups (PGs). The entire openEHR community can access the repository for retrieval, or “copy-out”. Those developers in identified project groups can perform modifications to the controlled items according to the process described below. All community members can raise Problem Reports, and those members in an openEHR team can raise Change Requests.

The key elements of this environment are as follows.

Repository
The repository of deliverables (centre). Includes all documents, software source, and related information needed to recreate a deliverable from scratch.

The Configuration Management (CM) system
The system controlling access to the repository, performs versioning of controlled items, release identification, and manages change requests. The CM system enables any previous version of the repository to be obtained. Implemented in openEHR using Subversion and various online change request and problem reporting tools.

Project Group (PG)
Formally constituted team that is responsible for the development of deliverables of the project. These teams can be considered “formal” developers in the sense that they are defined users in the CM
system and can execute a change via a check-out / modify / validate / check-in sequence. The project
group can raise Problem Reports (PRs) and Change Requests (CRs) at any time and are also responsible
for preliminary review of non-trivial PRs and CRs according to the change management process
described below.

The User Community
The openEHR community at large, consisting of any user or interested person or organisation. Users
in the community who are not in the informal or formal development pool, can copy-out all delivera-
bles and can raise PRs. Users who are not otherwise developers or technically involved in any way
typically only download binary software builds.

The Architectural Review Board (ARB)
The ARB is formally constituted of experts from diverse backgrounds, and operates according to the
openEHR ARB Terms of Reference. Its main activity is the review of major CRs, according to the
change management process described below. The ARB does not create PRs or CRs, and it does not
review PRs, being concerned only with change. (Naturally there may be some members of the ARB
who, in their role as a PG member may create PRs and CRs).

6.2 The Change Process
This section describes in detail the change process that applies to openEHR projects. However, read-
ers do not need to know all the details to work on a project - the following processes and documenta-
tion are generally supported by online tools that ensure that the process is easy to participate in and
follow.

6.2.1 Overview
Changes are made to a repository by members of the relevant project group. All changes have a
Change Request (CR). A CR can only be raised by someone in the project group, and is the key
document in the change process, being used to record all status and analysis information relating to
the change from its opening to rejection or resolution. CRs are raised either to fix problems, or to per-
form enhancements to a component. A CR that is designed to fix a problem may refer to existing PRs
(usually problems reported in released binaries by users), or the problem may simply be documented
in the CR itself (typically the case when a developer finds a problem).

A PR is raised describing in detail a problem or deficiency in a component or product, as perceived by
a user (including developers acting as testers). Such descriptions tend to be at a coarse granularity of
component or functionality, and only about main releases. A PR can be created by anyone. A PR cor-
responds to a “black-box” view of the product or component - the raiser doesn’t care how it is imple-
mented, only that it is not working properly. Problem Reports have a simple lifecycle: they are raised,
then either rejected or resolved. If not rejected, a PR is always resolved by one or more CRs. A CR
may resolve one or more PRs. FIGURE 7 illustrates the relationship between PRs and CRs within the
user and developer spaces respectively.

The scope of CR is always a whole repository (i.e. a whole project) even if it changes only a single
file. In many cases, a CR causes changes in one project (e.g. the specification project) that will need
to be taken into account in another repository (e.g. one of the implementation projects). It is up to the
managers of each project to decide on an appropriate moment to incorporate the relevant changes in
their repository.
6.2.2 Problem Reporting

New Problem Reports (PRs) are created by users. They are reviewed initially by the relevant project group, and are either rejected or cause the creation of one or more new CRs, or the modification of existing CR(s). Any CR related to a PR in this way should include the PR id in its problem_description. The CR then enters the process described below. If the CR is implemented and solves the problem, any PR(s) referred to in its problem_description section are progressed to the resolved state. FIGURE 8 illustrates the PR lifecycle.

6.2.3 Change Request Process for openEHR Reference Projects

This section describes a change process in which a board of review as well as the project group processes CRs. In openEHR, this process is applied to all reference projects, and any ad hoc projects requiring more disciplined change control.

6.2.3.1 Workflow

A new CR created by any project developer, and may be due the review of one or more PRs. The process of handling a CR in a project using a review board is illustrated by FIGURE 9. If a CR is not rejected at some point, it is eventually implemented, causing changes in the appropriate repository.

Assuming sufficient repository-wide quality controls are applied before a CR is closed (such as document review, guaranteeing that change source compiles, builds and runs, and so on), the repository is
always guaranteed to move from one self-consistent state to another self-consistent state - there are no inconsistent states. This also means that every version of the repository is the product of some initial state plus the application of a known list of CRs. In this way, the quality of the repository is maintained.

6.2.3.2 CR Lifecycle

Each CR follows the lifecycle illustrated in FIGURE 10. The lifecycle is effectively the CR-centric view of what happens during the change process shown in FIGURE 9. Most CRs proceed through the states initial, analysis, implementation, v_and_v (“verification and validation”) to completed. On the way, some may be rejected; others may require ARB analysis and approval before being allowed to proceed. Some CRs may be discovered to be unimplementable during implementation, which will lead to them being put back in the analysis state, from which they might be rejected. Occasionally a

FIGURE 9 openEHR Change Request Process for reference project
CR may be *superseded* by a more recent decision; this is reflected in the transitions leading to the superseded state. The V&V state is so named because it covers both the testing software and reviewing documents.

![CR Lifecycle Diagram](image)

**FIGURE 10** CR Lifecycle

When it is decided that a CR will be progressed, it also has to be decided which release the final changes are intended for, and in which release the changes will be made for testing. If the former is production release 1.5, the latter will be the corresponding development release, which must be in test-only mode.

### 6.2.3.3 Project Group-managed CRs

Many CRs are for trivial problems such as errors in documentation, incorrectly defined elements of specifications, or small software bugs. The CRs are managed by the relevant project group. The PG assigns the CR to someone (or someone self-nominates). **All further changes to the CR are undertaken by the person to whom the CR is assigned to.** The CR is analysed, and if the work to execute it is within the current resources of the PG, it can be carried out. If the work is greater, or it is realised that it is a more serious category of change, the CR is passed to the ARB, by setting owner=ARB. For CRs that remain with the project group, the process is as follows:

- Implementation is done in the test release indicated in the CR; when deemed complete, the state is set to v_and_v, and the changes are tested/reviewed. If rejected, the CR state reverts to implementation, and further changes are made, according to the test_outcome field.

- When implementation and verification is complete, the changes are promoted into the repository of the target release indicated in the CR.

### 6.2.3.4 Review Board-managed CRs

Any CR whose category is requirements or architecture, or for which the work to do the change is significant, is reviewed by the ARB. The review process is as follows:

- a CR normally has an initial problem description (that may refer to one or more PRs) and change description; there may also be the beginnings of an impact analysis.
the CR goes to the ARB with owner=ARB and state=analysis

- the ARB assigns from among its members someone to manage the CR. This person becomes responsible for progressing the CR through its lifecycle. All further changes to the CR are undertaken by the CR’s assignee.

- the ARB members review the CR, and propose changes to the change_description, impact_analysis, target_release and test_release. An estimate of time & resources for the work is done, either by the ARB, or by asking a relevant non-ARB person.

- The CR assignee makes changes based on the input, and sets the state to awaiting_approval;

- The ARB either:
  - approves the CR (by simple majority vote), in which case it is implemented, tested and the changes incorporated into the relevant repository; or
  - it proposes further changes. Such changes might include setting the target and/or test releases to be some later release, or an experimental one, in order to remove risk to established deliverables; or
  - it discovers that it cannot reach a consensus on the proposed changes as documented in the change_description (e.g. there might be a modelling issue) or impact_analysis (there might be disagreement on how the change will affect real systems). In this case, the ARB agrees to:
    * hold a physical meeting or telephone conference to resolve the issue;
    * co-opt expert assistance;
    * seek input from the community input

- The CR assignee is responsible for ensuring that by one means or another, the CR is progressed, either to the point where it will be implemented in some release, or else it is rejected.

- When it has been decided that implementation will occur, the owner field will be set to PG and the state to implementation. In the case of documents or specifications, this simply means that the changes will be made to the documents.

- Implementation is done by the relevant project group in the test release indicated in the CR; when deemed complete, the state is set to v_and_v, and the changes are reviewed by the ARB. If rejected, the CR state reverts to implementation, and further changes are made, according to the test_outcome field.

- When implementation and verification is complete, the changes are promoted into the repository of the target release indicated in the CR.

Note that CRs are visible to the openEHR community for open inspection online, with an indication of intended dates of resolution & how to communicate a response to the ARB.

### 6.2.4 Change Request Process for ad hoc Projects

This section describes a change process that is used on non-reference (typically smaller) projects, where no formal board of review exists.

#### 6.2.4.1 Workflow

A new CR created by any project developer, and may be due the review of one or more PRs. The process of handling a CR in a project with no review board is illustrated by FIGURE 9. If a CR is not rejected at some point, it is eventually implemented, causing changes in the appropriate repository.
6.2.4.2 CR Lifecycle

The CR lifecycle, illustrated in FIGURE 12, is similar to the review board case, except that all decisions are taken by the project group (PG).

6.2.4.3 CR Management

CRs in the no review board situation are all managed by the project group, using the same steps undertaken by the PG and ARB in the review board case.

FIGURE 11 openEHR Change Request Process with no review board
FIGURE 12 Simple CR Lifecycle
7 Tools

7.1 Overview

The following figure illustrates the tool environment supporting openEHR development projects.

![Tool Environment Diagram](image)

**FIGURE 13** Tool Environment of openEHR projects

7.2 Configuration Management System

The CM system is the primary tool that supports the change process. openEHR uses Subversion for configuration management. Detailed instructions for Subversion appear under the “development” link on the openEHR website. The master repositories for the openEHR projects are hosted on the main openEHR server at the CHIME department of University College London.
7.3  PR / CR Database

PRs and CRs are managed on a project by project basis, using customised instances of the Plone Issue Collector in a Plone/Zope database at openEHR.org.

7.4  Publishing/Distribution

Publishing of usable software packages, documents etc is achieved via the use of the Subversion repository oe_distrib. Its contents are the binary packages for each platform, as of each release and/or tag. The client / server nature of Subversion makes it easy for a user to obtain just what he/she wants, without being concerned with the fact that the entire repository might grow quite large.
8 CI Identification

Configuration items in openEHR repositories are generally known simply by their path. The CM tools automatically generate unique, immutable identifiers, which are reliable, regardless of where the file might be moved.

Documents
Documents can be identified using an identifier of the following form:

    document_id = <artifact_id>

Where the fields are defined as:

- **artifact_id**: id corresponding to the subject of document, e.g. “ehr_im” (EHR information model)

Example document identifiers are as follows:

- ehr_im
- common_im

Document source files will always have names independent of their version, and that might vary according to which tool is used to produce them. Documents may be stored and distributed in various file formats, e.g. Adobe PDF, HTML etc. File names of documents generated for dissemination are of the form:

    <document_id>.<extension>

Where it is more convenient (or it would cause problems with some tools), dots may be replaced by underscores in version numbers. Examples include:

- ehr_im.pdf -- Adobe PDF file
- common_im.html -- HTML file

Computable Artifacts
All computable artifacts whether abstract or derived are identified by a file name of the form:

    <artifact_id>.<extension>

Neither the project id nor the version id are incorporated in the identifier. The former is redundant in such files, while the latter prevents automatic replacement of a previous version by a later version.

Examples include:

- ehr_im.idl -- IDL file
- common_im.xmi -- XMI file
- datatypes_am.xsd -- X-schema file

Programming language files will almost always be named according to a class name or something similar.
Appendix A  Forms

A.1  Problem Report Form

openEHR PROBLEM REPORT

ID <pr_id>    Date Raised: <date>

>Title>

RAISER: <person>    STATE: <state>

*PRIORITY: <priority>  SEVERITY: <severity>

[RELATED_CRs: <related CRs> ]

PROBLEM DESCRIPTION

COMPONENT: <component_id>

PROBLEM_DESCRIPTION: <text>

RESOLUTION

Date Closed: <date>

RESOLUTION_DESCRIPTION: <text>

NOTES: <notes>

A.2  PR Form Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr_id</td>
<td>id of form “PR_nnnnn”</td>
</tr>
<tr>
<td>date_raised</td>
<td>yyyy-MM-dd</td>
</tr>
<tr>
<td>title</td>
<td>text</td>
</tr>
<tr>
<td>raiser</td>
<td>name &lt;email address&gt;</td>
</tr>
<tr>
<td>state</td>
<td>opened, review, resolved, rejected</td>
</tr>
<tr>
<td>priority</td>
<td>Assigned by openEHR; values from 1, 2, 3</td>
</tr>
<tr>
<td>severity</td>
<td>critical, high, moderate, low</td>
</tr>
<tr>
<td>related CRs</td>
<td>List of CR ids for CRs</td>
</tr>
<tr>
<td>component_id</td>
<td>identifiers of released component manifesting problem</td>
</tr>
<tr>
<td>problem_description</td>
<td>text - problem as perceived by user</td>
</tr>
<tr>
<td>date_closed</td>
<td>yyyy-MM-dd</td>
</tr>
<tr>
<td>resolution_description</td>
<td>text</td>
</tr>
<tr>
<td>notes</td>
<td>text</td>
</tr>
</tbody>
</table>
A.3 Change Request Form

openEHR CHANGE REQUEST

ID <cr_id>          Date Raised: <date>

RAISER: <person>

OWNER: <CMS | PG | ARB>           STATE: <state>
ASSIGNED_TO: <person>

PROBLEM DESCRIPTION: [ text | <list of PR ids>]
[Dependencies: <list of CR ids>]

CHANGE DESCRIPTION

CATEGORY: <category: documentation | error | design | requirements>

IMPACT ANALYSIS: <text>
ANALYST: <person>
CHANGE DESCRIPTION: <text>

COMPONENTS AFFECTED:
<change_component, version>
...

APPROVED by: <person>
IMPLEMENTOR: <person>

TARGET Release: <baseline id>

VERIFICATION & VALIDATION

TEST Baseline: <baseline id>
Test Outcome: <text>

RESOLUTION

Date Closed: <date>
[Reason for Rejection: <text>]

NOTES: <notes>

A.4 CR Form Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>CR_nnnnnn</td>
</tr>
<tr>
<td>dateRaised</td>
<td>yyyy-MM-dd</td>
</tr>
<tr>
<td>title</td>
<td>text</td>
</tr>
<tr>
<td>raiser</td>
<td>name &lt;email address&gt;</td>
</tr>
<tr>
<td>Field</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>owner</td>
<td>CMS (config mgt system), PG (project group), ARB</td>
</tr>
<tr>
<td>assigned_to</td>
<td>name of ARB member responsible for progressing the CR</td>
</tr>
<tr>
<td>state</td>
<td>initial, rejected, PG_analysis, ARB_analysis, implementation, v_and_v, completed, superseded</td>
</tr>
<tr>
<td>problem description</td>
<td>text description, or else reference to list of ids of PRs generating this CR</td>
</tr>
<tr>
<td>dependencies</td>
<td>list of ids of CRs whose completion is required for the completion of this CR</td>
</tr>
<tr>
<td>category</td>
<td>documentation, error, design, requirements</td>
</tr>
<tr>
<td>impact_analysis</td>
<td>text describing impact on rest of release</td>
</tr>
<tr>
<td>analyst</td>
<td>name &lt;email address&gt;</td>
</tr>
<tr>
<td>change_description</td>
<td>text describing what should be changed</td>
</tr>
<tr>
<td>items affected</td>
<td>list of items</td>
</tr>
<tr>
<td>approved by</td>
<td>name &lt;email address&gt;</td>
</tr>
<tr>
<td>implementor</td>
<td>name &lt;email address&gt;</td>
</tr>
<tr>
<td>target release</td>
<td>release by which this CR must be resolved</td>
</tr>
<tr>
<td>test baseline</td>
<td>release in which changes due to this CR will first appear for testing</td>
</tr>
<tr>
<td>test outcome</td>
<td>either “passed”, or a reason for test failure</td>
</tr>
<tr>
<td>date closed</td>
<td>date on which CR was completed</td>
</tr>
<tr>
<td>reason for rejection</td>
<td>text</td>
</tr>
</tbody>
</table>
END OF DOCUMENT
openEHR / ISO 18308 Conformance Statement

Authors: T Beale¹

Revision: 1.5

Pages: 51

¹. Ocean Informatics Australia

© 2002-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>Updated to Release 1.0 reference model semantics.</td>
<td>T Beale</td>
<td>02 Feb 2006</td>
</tr>
<tr>
<td>1.4</td>
<td>Updated to draft openEHR 0.9 Release for HealthConnect Australia. Changed title of document.</td>
<td>T Beale</td>
<td>20 Mar 2003</td>
</tr>
<tr>
<td>1.3</td>
<td>Updated to EHR RM 3.8.2 and Data Types RM 1.5.8; ISO Requirements TS V1.0</td>
<td>T Beale</td>
<td>10 Nov 2002</td>
</tr>
<tr>
<td>1.2</td>
<td>Updated to EHR RM 3.5 and Data Types RM 1.52</td>
<td>T Beale</td>
<td>19 Aug 2002</td>
</tr>
<tr>
<td>1.11</td>
<td>Minor adjustments.</td>
<td>T Beale</td>
<td>5 Jul 2002</td>
</tr>
<tr>
<td>1.1</td>
<td>Updated to latest ISO requirements draft.</td>
<td>T Beale, A Goodchild</td>
<td>28 Jun 2002</td>
</tr>
<tr>
<td>1.0</td>
<td>Initial Writing</td>
<td>T Beale</td>
<td>20 Jun 2002</td>
</tr>
</tbody>
</table>

Acknowledgements

The work reported in this paper has been funded by a number of organisations, including The University College, London; Ocean Informatics, Australia.
Table of Contents

1 Introduction................................................................. 7
  1.1 Purpose............................................................................. 7
  1.2 Status............................................................................... 7
    1.2.1 Versions................................................................. 7
    1.2.2 Completeness......................................................... 7
  1.3 Acronyms, Abbreviations and Definitions ....................... 8
  1.4 Methodology............................................................... 8
  1.5 Recommendations...................................................... 9

2 Mappings.............................................................................. 11
  ISO Section 1 Structure .................................................. 11
    ISO Section 1.1 Record organisation ................................ 11
    ISO Section 1.1.1 Sections .............................................. 11
    ISO Section 1.1.2 EHR format .......................................... 11
    ISO Section 1.1.3 Portability ........................................... 12
    ISO Section 1.1.4 Secondary uses .................................... 12
    ISO Section 1.1.5 Archiving ........................................... 12
    ISO Section 1.2 Data Organisation .................................... 12
    ISO Section 1.2.1 Structured Data .................................... 13
    ISO Section 1.2.2 Non-structured data ............................... 13
    ISO Section 1.2.3 Clinical Data ........................................ 15
    ISO Section 1.2.4 Administrative data ............................... 15
    ISO Section 1.3 Type and form of data ............................... 16
    ISO Section 1.3.1 Support for different types of data .......... 17
    ISO Section 1.3.2 Data types ........................................... 17
    ISO Section 1.3.3 Reference data ..................................... 19
    ISO Section 1.3.4 Contextual Data .................................... 19
    ISO Section 1.3.5 Links .................................................. 20
    ISO Section 1.4 Supporting health concept representation ...... 20
    ISO Section 1.4.1 Support for multiple coding systems ......... 20
    ISO Section 1.4.2 Unique representation of information ........ 21
    ISO Section 1.4.3 Language independence ............................ 21
    ISO Section 1.4.4 Representation of text ............................ 21
  ISO Section 2 PROCESS .................................................... 22
    ISO Section 2.1 Clinical processes ..................................... 22
    ISO Section 2.1.1 Support for clinical processes ................. 22
    ISO Section 2.1.2 Problems/issues and health status .......... 23
    ISO Section 2.1.3 Clinical reasoning .................................. 23
    ISO Section 2.1.4 Decision support, guidelines, and protocols .. 24
    ISO Section 2.1.5 Care Planning ....................................... 25
    ISO Section 2.1.6 Orders & service processes ........................ 25
    ISO Section 2.1.7 Integrated care ..................................... 26
    ISO Section 2.1.8 Quality assurance ................................... 26
    ISO Section 2.2 Record processes ..................................... 26
    ISO Section 2.2.1 Data capture ........................................ 26
    ISO Section 2.2.2 Retrieval/query/views of data .................... 27
    ISO Section 2.2.3 Presentation of data .............................. 28
ISO Section 2.2.4 Scalability ........................................... 28

ISO Section 3 COMMUNICATION .................................... 29
ISO Section 3.1 Messaging ........................................... 29
ISO Section 3.2 Record exchange .................................. 30

ISO Section 4 PRIVACY AND SECURITY .................... 31
ISO Section 4.1 Privacy and confidentiality .................... 31
ISO Section 4.2 Consent ............................................ 31
ISO Section 4.3 Access control .................................... 32
ISO Section 4.4 Data integrity .................................... 32
ISO Section 4.5 Auditability of access .......................... 33

ISO Section 5 MEDICO-LEGAL ................................. 34
ISO Section 5.1 Support for legal requirements ................ 34
ISO Section 5.2 Actors ............................................. 34
ISO Section 5.2.1 Subject of healthcare ....................... 34
ISO Section 5.2.2 Patient identification ...................... 35
ISO Section 5.2.3 User Identification ......................... 35
ISO Section 5.2.4 Healthcare parties ......................... 35
ISO Section 5.2.5 Author responsibility ...................... 36
ISO Section 5.2.6 Attestation/Authorization of entries .... 36
ISO Section 5.3 Clinical competence/governance .......... 37
ISO Section 5.4 Faithfulness ..................................... 37
ISO Section 5.5 Preservation of context ...................... 37
ISO Section 5.6 Permanence ...................................... 38
ISO Section 5.7 Version control .................................. 38

ISO Section 6 ETHICAL ........................................... 39
ISO Section 6.1 Support for ethical justification ............ 39

ISO Section 7 CONSUMER/ CULTURAL .................... 40
ISO Section 7.1 Consumer issues ............................. 40
ISO Section 7.1.1 Support for consumer issues ............. 40
ISO Section 7.2 Cultural issues ................................ 41
ISO Section 7.2.1 Support for cultural issues .............. 41

ISO Section 8 EVOLUTION ...................................... 42
ISO Section 8.1 Support for EHR architecture and EHR system evolution 42

3 Summary of Conformance Exceptions ....................... 45

4 ISO Requirements Requiring Review ......................... 47

A References .................................................................. 49
1 Introduction

1.1 Purpose

This document describes the compliance of the openEHR architecture to the draft ISO TC 215 Technical Specification ISO/WD 18308, “Requirements for an Electronic Health Record Reference Architecture”. These requirements have been developed from numerous sources. Quoting from the 2002-06-28 draft:

An extensive search of the literature and direct contact with domain experts in many countries has been undertaken to identify as many existing sources of EHR requirements as possible. Material from over 35 primary sources has been obtained. This includes 20 sources originally collected by the EHCR Support Action Project (EHCR-SupA) in Europe. This project was established to support the work of CEN in developing a four part EHR communication standard (CEN 13606, 1999) and one of its Deliverables [SupA.1.4, 2000] was to provide “...a consolidated classification of the requirements for the Electronic Health Care Record (EHCR) and EHCR architecture (EHCRA).” The 20 different primary EHR requirements documents used by EHCR-SupA came from many sources including relevant projects from the EU’s Third and Fourth Framework AIM programmes and from CEN. The 15 newly identified sources come from the United States, The Netherlands, Australia, and New Zealand.

As implied by the title, the ISO EHR requirements relate to EHR “reference architectures”, which include the architectures such as that published by openEHR. It is therefore appropriate to show how the openEHR architecture satisfies or deviates from the the ISO requirements.

1.2 Status

1.2.1 Versions

This document compares the ISO requirements draft document identified as ISO/WD 18308, published by ISO TC 215 WG1, dated 2002-06-28 with the openEHR deliverables:

- openEHR EHR Reference Model (RM) revision 4.3 draft
- openEHR EHR Demographic Reference Model revision 1.4.1 draft
- openEHR EHR Common Reference Model revision 1.4.3
- openEHR EHR Data Structures revision 1.3.1
- openEHR EHR Data Types revision 1.7.2
- openEHR archetype draft documents corresponding to these RM versions

In the future it is expected that this list will be expanded to include the following document:

- openEHR consent/policy reference model

This document is available at http://svn.openehr.org/specification/TAGS/Release-1.0/publishing/requirements/iso81308_conformance.pdf.

The latest version of this document can be found at http://svn.openehr.org/specification/TRUNK/publishing/requirements/iso81308_conformance.pdf.

1.2.2 Completeness

In the conformance column of the tables, the systems listed as having verified each feature in the architecture are not exhaustive, and undoubtedly do not include systems which have in fact verified the feature. These can be added over time.
1.3 Acronyms, Abbreviations and Definitions

For the most part, definitions of EHR-related concepts are to be found in ISO/WD 18308, which includes a comprehensive set of definitions. Abbreviations used in describing conformance are described below. A few key acronyms are repeated here for convenience.

- **EHR** - Electronic Health Record. There is currently no single definition of “EHR” in ISO;
- **EHRRA** - EHR Reference Architecture, i.e. a formal model of EHR semantics as derived from a set of requirements, and containing no design particularities, or features specific to any jurisdiction, style of medicine, or culture;
- **RM** - reference model; any formal model derived from requirements by analysis, but prior to the application of system design activities.

1.4 Methodology

The approach taken in this document is to show for each ISO requirement what feature(s) of the openEHR reference architecture satisfy the requirement.

The information is presented in a table of the form shown below. All ISO original content is shown in blue. All ISO numbering is preserved. Each requirement includes a number in parentheses denoting the heading number in a heading framework developed by ISO WG1 for the purpose of classifying requirements. The numbers refer to version 5.3 of the heading framework.

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDN.N</td>
<td>ISO description, verbatim from the current ISO draft document. (ISO source reference number)</td>
<td>Detail of how openEHR meets the requirement</td>
<td>Level of conformance (see below).</td>
</tr>
</tbody>
</table>

Conformance

Conformance is described in two dimensions. The conformance of the openEHR models to the requirement is indicated by paragraphs like the following in the “conformance” column.

**Design: X**

The following values of the letter ‘X’ are used:

- **Full**: Full conformance - the requirement is believed to be completely satisfied by the reference architecture in a direct way, e.g. with a class or other feature specifically designed for the purpose;
- **Qual**: Qualified - the requirement is believed to be completely satisfied by the architecture;
- **Part**: Partial - the requirement is partially met by the current architecture;
- **Fut**: Future - the requirement will be met by a future revision of the architecture;
- **No**: No conformance - the requirement is not met by the architecture and is not intended to be satisfied in the future;
- **N/A**: Not Applicable - In some cases, the current ISO requirement is not considered a valid requirement for EHR reference architectures;
- **Unk**: Unknown - it is currently not known if the architecture caters for the requirement, or if it does it in a manner desirable for implementation and information management.

Design conformance essentially indicates whether, in the openEHR design process, there has been conscious consideration of the ISO requirement or one which is very similar or a superset, or use cases which are implied by the ISO requirement.

However, the best intentions of design do not always guarantee success in implemented systems, due to factors...
such as complexity, novelty (never before implemented), or difficulty of testing (e.g. requires large clinical trials). Hence, the second dimension indicates whether the openEHR reference architecture feature, or one like it (e.g. in one of the architectures on which openEHR is based, such as CEN 13606, GEHR or SynEx) has been shown to work in practice. This is shown by “Validated” paragraphs like the following.

**Val: xxxx**

The values here include the names of any of the following projects where a) the openEHR design feature was present and b) it was known to have fulfilled the ISO requirement.

- **CEN:** Any CEN ENV 13606 implementation
- **GEHR:** The Australian Good Electronic Health Record project [9]. (It should be noted that the Australian GEHR project (1997 - 2002) was heavily implementation oriented, while the original Good European Health Record project (1992 - 1995) was a requirements-oriented project, and one of the precursors to the ISO 18308 technical specification described here);
- **HL7:** ....
- **OMG CorbaMed (HDTF):** ...
- **SNX:** SynEx & Synapses European projects [4], [5]

The conformance assessments provide a guide to what elements of the openEHR architecture need to be addressed in order to meet the ISO requirements. Non-conforming requirements are summarised in a hyperlinked list at the end of this document.

### 1.5 Recommendations

Three general rules of thumb should be respected by any requirement in a set of requirements, as follows:

- each statement expresses one requirement only;
- it is clear how each statement would be tested, i.e. it would be easy to write a test case corresponding to the requirement;
- each statement expresses a requirement about the object of the requirements, not about something else (usually related).

The current version of the ISO requirements does not always follow these rules. Some requirements (e.g. 3.9, 3.15) actually express several requirements, and it is recommended that these be split out. Any requirement where multiple “design” entries are included in the conformance column.

Some requirements are unclear or vague, and the meaning is not obvious, nor is a way to state a test case. Finally, in a few cases, a few requirements are not considered to apply to an EHR reference architecture. In both cases these are indicated with a “TBR” (to be reviewed) paragraph of the following form:

**TBR 1: example TBR paragraph**

These paragraphs may indicate the need for further review of the ISO requirement, either within ISO TC 215 (in order to state the requirement more clearly, or correct it) or within openEHR, in order to better interpret the requirement. A hyperlinked summary list of TBR paragraphs is provided at the end of this document.
## 2 Mappings

### ISO Section 1 Structure

#### ISO Section 1.1 Record organisation

#### ISO Section 1.1.1 Sections

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR1.1</td>
<td>The EHRRA must enable information in the EHR to be organised in different sections allowing navigation by users and views of sections to be returned as the result of queries. (1.1)</td>
<td>openEHR EHR RM: COMPOSITIONs provide coarse-grained buckets. FOLDERs reference COMPOSITIONs, providing course-grained views.</td>
<td>Design: Full Val: GEHR, CEN</td>
</tr>
</tbody>
</table>

### ISO Section 1.1.2 EHR format

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR1.2</td>
<td>The EHRRA must ensure that the 'format' of the EHR as it appears to the clinician or user is able to conform to specifications set by standards organisations, regulatory and accreditation agencies, professional groups, local healthcare institutions and users. (1.1)</td>
<td>openEHR EHR RM: The reference model architecture provides generic structures which are configured by domain-authored archetypes, which can be devised to reflect relevant standards in structuring.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
ISO Section 1.1.3  Portability

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR1.3</td>
<td>The EHRRA must support an EHR which is moveable and mergeable between individuals and institutions independent of hardware, software (application programs, operating systems, programming languages), databases, networks, coding systems, and natural languages. (2.6)</td>
<td>openEHR all models: All models are defined in platform-independent UML, and can be expressed in any object-oriented formalism for implementation, including XML-schema or other XML schema languages (RDF, Schematron etc). Moving and merging semantics are defined by the VERSIONED_COMPOSITION and EHR_EXTRACT classes.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

ISO Section 1.1.4  Secondary uses

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR1.4</td>
<td>The EHRRA must enable information in the EHR to be organised and retrieved in a manner that facilitates its secondary uses. (1.1)</td>
<td>openEHR EHR RM: Paths, FOLDERS, COMPOSITIONs, and SECTIONs all provide means of retrieving EHR data in arbitrary ways.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

ISO Section 1.1.5  Archiving

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR1.5</td>
<td>The EHRRA must support archiving (5.4)</td>
<td>openEHR EHR RM: The ‘contribution’ concept, implemented with versioned COMPOSITIONs which contain a AUDIT_DETAILS for each change enable each successive change to the EHR to be unambiguously identified; each change can therefore be retrieved and processed by an archiving system.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

ISO Section 1.2  Data Organisation
## ISO Section 1.2.1 Structured Data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR2.1</td>
<td>The EHRRA must enable storage of data as lists such that the order of the data is preserved when the data is displayed. (1.2.1)</td>
<td>openEHR Data Structures RM: ITEM_LIST subtype of ITEM_STRUCTURE.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>STR2.2</td>
<td>The EHRRA must enable storage of data in tables such that the relationships of the data with the row and column headings are preserved. (1.2.1)</td>
<td>openEHR Data Structures RM: ITEM_TABLE subtype of ITEM_STRUCTURE.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>STR2.3</td>
<td>The EHRRA must enable storage of data in hierarchies such that the relationship between the node parents and children are preserved. (1.2.1)</td>
<td>openEHR Data Structures RM: ITEM_TREE subtype of ITEM_STRUCTURE.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>STR2.4</td>
<td>The EHRRA must enable storage of data such that simple name/value pairing is preserved. (1.2.1)</td>
<td>openEHR Data Structures RM: ITEM_SINGLE subtype of ITEM_STRUCTURE.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>STR2.5</td>
<td>The EHRRA must enable the storage of multiple values of the same measurement taken at closely proximate times at the same contact, or at different contacts and at different locations. The context of these measurements must be preserved - such as who took the measurement, what method was used etc. These values should be able to be returned in a query and ordered in different ways. (1.1)</td>
<td>openEHR Data Structures RM: classes HISTORY&lt;T:ITEM_STRUCTURE&gt;, POINT_EVENT, INTERVAL_EVENT, etc enable recording of time-series data of any complexity, along with their times. Other context data is recorded on the owning OBSERVATION. Recordings by different people, using different protocols etc are not considered scientific time-series data due to variability of samples, and are recorded using successive, distinct ENTRYs.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

## ISO Section 1.2.2 Non-structured data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR2.6</td>
<td>The EHRRA must support the inclusion of narrative free text and there should be no logical limit to the size of this text. (1.2.2.1)</td>
<td>openEHR Data Types: DV_TEXT and DV_PARAGRAPH types for plain or text with basic font formatting; DV_ENCAPSULATED for encapsulated rich text.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR2.7</td>
<td>The EHRRA must support searching within non-structured data (text and non-text) and the inclusion of structured text within this data. (1.2.2.1)</td>
<td>openEHR Data Types: Searching is not strictly an EHR reference model facility - it can be performed on any DV_TEXT or DV_PARAGRAPH item, as converted to e.g. XML or any other database or representation format. The DV_PARAGRAPH type can include any number and mixture of DV_CODED_TEXTs (coded terms) and DV_TEXTs.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>ISO Req’t</td>
<td>Description</td>
<td>openEHR artifact</td>
<td>Conformance</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>STR2.8</td>
<td>The EHRRA must support the inclusion of comments within the data stored - enabling the clinician to qualify structured information appropriately. Comments must be able to be linked to specific data attributes. (1.2.2.2)</td>
<td><em>openEHR Reference Model:</em> Comments are expressed as text data items in distinct <code>ELEMENTs</code> in structured data. Associating a comment with a specific datum means using archetypes to define the relevant structure (e.g. <code>ITEM_TREE</code> etc) to allow comments to be associated with original data items.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>STR2.9</td>
<td>The EHRRA must provide a means for different levels of emphasis to be associated with comments and other entries - this may alter the way they are displayed or their returning in a query. (1.2.2.2)</td>
<td>The <code>DV_TEXT</code> type provides a facility to associate a platform-standard font string with one or more text items.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>
### ISO Section 1.2.3  Clinical Data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
</table>
| STR2.10   | The EHRRA must allow for comprehensive information storage and retrieval regarding patient care. The EHRRA must at a minimum allow for the recording of all data on:  
- Patient history  
- Physical examination  
- Psychological, social, environmental, family, and self care information  
- Allergies and other therapeutic precautions  
- Preventative and wellness measures such as vaccinations and lifestyle interventions  
- Diagnostic tests and therapeutic interventions such as medications and procedures  
- Clinical observations, interpretations, decisions, and clinical reasoning  
- Requests/orders for further investigation, treatments, or discharge  
- Problems, diagnoses, issues, conditions, preferences and expectations  
- Healthcare plans, health and functional status, and health summaries  
- Disclosures and consents  
- Suppliers, model and manufacturer of devices (e.g. implants or prostheses) | openEHR EHR RM:  
All of these categories except “Disclosures and consents” would be recorded by the normal means in COMPOSITIONs (mostly “persistent” COMPOSITIONs), which can be thematically defined as required.  
“Disclosures” are not strictly part of the openEHR EHR model, but would be recorded in the EHR system (i.e. openEHR has no particular model for how disclosures are recorded).  
“consents” are expressed using instances of the INSTRUCTION ENTRY subtype, and may also be further used by an access control service in mediating access to the EHR. | Design: Qual |

### ISO Section 1.2.4  Administrative data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
</table>
| STR2.11   | The EHRRA must support the recording (and classifying for identification purposes) of patient identification, location, demographic, contact, employment and other administrative data. (1.3.3) | openEHR Demographic RM:  
The demographic model defines the class PARTY and various subtypes, all archetypable to whatever particular form is required. | Design: Full  
Val: CEN, GEHR, SNX |
<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR2.12</td>
<td>The EHRRA must support standards for information which enable the unambiguous identification of the subject of care, the clinicians involved in care (including their role and context of care), the location of care, the date/time and duration of care, and third parties such as next of kin and non-clinical contacts. There should be no limit on the storage of such information. (1.3.3)</td>
<td><em>openEHR Demographic RM:</em> The demographic model defines the class <code>PARTY</code> and various sub-types, all archetypable to whatever particular form is required. <em>openEHR EHR RM:</em> The class <code>EVENT_CONTEXT</code> and the context attributes of the <code>ENTRY</code> class define all of the attributes mentioned. Both classes can include unlimited further context data, with meaning defined by archetyping.</td>
<td>Design: Full  Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR2.13</td>
<td>The EHRRA must support the administration of healthcare processes and episodes of care as well as the organisation of visit and encounter data. (1.3.3)</td>
<td><em>openEHR EHR RM:</em> Episodes can be represented using <code>FOLDERs</code> for grouping all event compositions which occur during an episode. Encounters are represented using “event” <code>COMPOSITIONs</code>.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR2.14</td>
<td>The EHRRA must support the recording of financial and other commercial information such as health plan enrolment, eligibility and coverage information, guarantor, costs, charges, and utilisation. (1.3.3)</td>
<td><em>openEHR EHR RM:</em> All of these items can be recorded as persistent Compositions using appropriately designed archetypes, although in distributed systems (and particularly outside North America), this information is more likely to be in other systems in the environment.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR2.15</td>
<td>The EHRRA must support the recording of legal status and consents relevant to the patient’s healthcare (e.g. legal status of guardianship order, consents for operations and other procedures).</td>
<td><em>openEHR EHR RM:</em> All such details can be modelled using archetypes which describe the subject of the record. The data should probably be recorded in a dedicated persistent Composition.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR2.16</td>
<td>The EHRRA must be amenable to querying for the purpose of data aggregation to support information gathering required for population and public health initiatives, surveillance, and reporting.</td>
<td><em>openEHR EHR RM &amp; archetypes:</em> Archetypes provide the basis for formulating intelligent queries. As long as information which is of interest in population queries (e.g. lifestyle, chronic disease etc) has been stored using archetypes in the first place, very efficient querying is possible, based on the use of paths extracted from archetypes.</td>
<td>Design: Qual</td>
</tr>
</tbody>
</table>

### ISO Section 1.3  Type and form of data
## ISO Section 1.3.1  Support for different types of data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 3.1</td>
<td>The EHRRA must allow for the incorporation of data types defined elsewhere, such as DICOM, MIME, EKG. (1.3.1)</td>
<td>openEHR Data Types: The DV_ENCAPSULATED type caters for all data types defined in other standards.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

## ISO Section 1.3.2  Data types

The EHRRA must define the following data types:

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 3.2</td>
<td>Numeric and Quantifiable data. The EHRRA must support the definition of the logical structure of numeric and quantifiable data, including the handling of units. (1.3.4.2)</td>
<td>openEHR Data Types: DV_QUANTITY (including units), DV_CUSTOMARY_QUANTITY, DV_DATE/TIME types</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 3.3</td>
<td>Quantities should include a measure of precision related to the method of measurement. (1.3.4.2)</td>
<td>openEHR Data Types: precision is included as an attribute in the type DV_QUANTITY. More complex measurement information can be included in the OBSERVATION.protocol attribute which is of type ITEM_STRUCTURE (i.e. any complexity)</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 3.4</td>
<td>Percentages must be able to be expressed as quantities. (1.3.4.2)</td>
<td>openEHR Data Types: Percent is a valid unit in the Unified Code for Units of Measure (UCUM) specification [7], which provides the semantics for units in the DV_QUANTIFIED types.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 3.5</td>
<td>Quantity ranges The EHRRA must support the definition of the logical structure of ranges - that is high and low values. (1.3.4.2)</td>
<td>openEHR Data Types: Ranges are provided for with the DV_INTERVAL&lt;T:DV_ORDERED&gt; type, which caters for ranges of any ordered data types.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 3.6</td>
<td>Quantity ratios The EHRRA must support the definition of the logical structure of quantity ratios (i.e. x of a per y of b). (1.3.4.2)</td>
<td>openEHR Data Types: The DV_QUANTITY_RATIO type provides for ratios of the form &lt;{val_1, units_1}/{val_2, units_2}&gt;. What the quantities are of can be recorded in the name or in an associated attribute, but are not recorded inside the quantity ratio data item as such.</td>
<td>Design: Qual Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>ISO Req’t</td>
<td>Description</td>
<td>openEHR artifact</td>
<td>Conformance</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>STR 3.7 Dates and times</td>
<td>The EHRRA must support the definition of the logical structure of dates and times. (1.3.4.3)</td>
<td><em>openEHR Data Types:</em> The subtypes of <code>DV_CUSTOMARY_QUANTITY</code> provide for dates and times, namely <code>DV_DATE</code>, <code>DV_TIME</code>, <code>DV_DATE_TIME</code>, <code>DV_PARTIAL_DATE</code>, <code>DV_PARTIAL_TIME</code>, <code>DV_DURATION</code>.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
</tbody>
</table>
| STR 3.8 The EHRRA must support approximate, partial, and fuzzy dates and times such as: | | *openEHR EHR RM / Data types:* These requirements are satisfied with the following elements of the `openEHR` models:  
  - text data types  
  - `DV_PARTIAL_DATE` | Design: Qual Design: Full |
| STR 3.9 The EHRRA must support the recording of future planned events or actions such as: | | *openEHR EHR RM / Data types:* These requirements are satisfied with the following elements of the `openEHR` models:  
  - `DV_INTERVAL<>` of any date/time type  
  - `DV_TIME_SPECIFICATION` type  
  - `DV_TIME_SPECIFICATION` (with event alignment)  
  - `DV_TIME_SPECIFICATION`. One week after last dose: `HISTORY<T>` with reference event set to “last dose”  
| STR 3.10 The EHRRA must support the recording of time as an absolute time, an elapsed time since a particular event, and as a duration. (1.3.4.3) | | *openEHR Data types:* Absolute time: `DV_DATE_TIME`; elapsed time: `DV_DURATION`.  
*openEHR EHR RM:* `HISTORY<T>` allows events to be recorded with respect to a reference even. | Design: Full Val: CEN, GEHR, SNX |
| STR 3.11 The EHRRA must support the recording of the time-zone in which the recording took place. (1.3.4.3) | | *openEHR Data types:* Timezone is an attribute of `DV_DATE_TIME` `DV_DATE`, and `DV_TIME`. | Design: Full |
| STR 3.12 The EHRRA must support recording of time in all units down to milliseconds. (1.3.4.3) | | *openEHR Data types:* All date/time types support milliseconds. | Design: Full |
### ISO Section 1.3.3  Reference data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 3.13</td>
<td>The EHRRA must support the recording of references such as normal ranges and attributes relevant to a particular observation or measurement. (1.3.5)</td>
<td>openEHR Data types: DV_ORDERED, reference_ranges and normal_range attributes.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

### ISO Section 1.3.4  Contextual Data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 3.14</td>
<td>The EHRRA must support the recording of contextual data associated with the date/time the event occurred.</td>
<td>openEHR EHR RM: EHR RM attribute HISTORY.origin, EVENT.offset</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.15</td>
<td>The EHRRA must support the recording of contextual data associated with the date/time the event was committed to the record.</td>
<td>openEHR EHR RM: EHR RM attribute AUDIT_DETAILS.time_committed</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.16</td>
<td>The EHRRA must support the recording of contextual data associated with the subject.</td>
<td>openEHR EHR RM: EHR RM attribute ENTRY.subject</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.17</td>
<td>The EHRRA must support the recording of contextual data associated with the person responsible for recording and committing the event.</td>
<td>openEHR EHR RM: EHR RM attribute ENTRY.provider (= information provider) EHR RM attribute EVENT_CONTEXT.composer EHR RM attribute AUDIT_DETAILS.committer</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.18</td>
<td>The EHRRA must support the recording of contextual data associated with the healthcare facility.</td>
<td>openEHR EHR RM: EHR RM attribute EVENT_CONTEXT.healthcare_facility</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.19</td>
<td>The EHRRA must support the recording of contextual data associated with the location where the event was recorded.</td>
<td>openEHR EHR RM: EHR RM attribute EVENT_CONTEXT.location</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 3.20</td>
<td>The EHRRA must support the recording of contextual data associated with the reason for recording the information associated with the event.</td>
<td>openEHR EHR RM: EHR RM attribute CARE_ENTRY.guideline_id</td>
<td>Design: Partial</td>
</tr>
<tr>
<td>STR 3.21</td>
<td>The EHRRA must support the recording of contextual data associated with the protocol associated with the event.</td>
<td>openEHR EHR RM: EHR RM attribute CARE_ENTRY.protocol</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
## ISO Section 1.3.5  Links

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 3.22</td>
<td>The EHRRA must define the semantic representation of links between different information in the EHR. (1.3.7)</td>
<td>openEHR Data types: LINK data type, including meaning attribute.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 3.23</td>
<td>The EHRRA must support links to ‘externally referenced data’ which is not able to be stored within the EHR, providing patient safety is not compromised. (1.3.7)</td>
<td>openEHR EHR RM: The DV_EHR_URI type can reference any complex data (e.g. demographic, terminological) in external repositories. openEHR Data Types: DV_ENCAPSULATED can include a URL for its data item which is not included by value in the EHR; DV_TEXT can include a URL as a hyperlink for a section of narrative text.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

## ISO Section 1.4  Supporting health concept representation

**ISO Section 1.4.1  Support for multiple coding systems**

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 4.1</td>
<td>The EHRRA must support multiple coding systems (entry or interface terminologies, reference terminologies and classifications) by creating interfaces with electronic tools such as terminology browsers, terminology editors and terminology servers. (1.4.1)</td>
<td>The openEHR models do this in several ways. Coded terms in EHR data are represented using the DV_CODED_TEXT type, which records the identity of the terminology from which codes come, using a TERMINOLOGY_ID object. The ids and codes refer to indentifiers and terms in the openEHR Terminology Model, which models the minimum semantics of the interface between the EHR and terminologies.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 4.2</td>
<td>At the data attribute level, the EHRRA must support the capture of the code, the coding scheme (e.g., coding/classification system), version and original language.</td>
<td>openEHR Data Types: The DV_CODED_TEXT &amp; CODE_PHRASE types record code, rubric (textual expansion) and terminology_id. Original language is recorded at the COMPOSITION level in AUDIT_DETAILS, because Compositions are not allowed to be of mixed language.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>STR 4.3</td>
<td>The EHRRA must enable storage of data from terminologies and preserve the information about the terminology set from which it was chosen (see section 1.4 below). (1.2.1)</td>
<td>openEHR Data Types &amp; EHR RM: CODE_PHRASE.terminology_id: TERMINOLOGY_ID.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
### ISO Section 1.4.2  Unique representation of information

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 4.4</td>
<td>Where information is not represented uniquely in only one place and one way, the EHRRA shall support explicit rules to avoid ambiguity (e.g. is must be clear what [not] [pedal pulses absent] means).</td>
<td>openEHR archetypes provide the semantic definition of data, including variant ways of recording the same data. Negation in particular is recorded using an “exclusion” archetype.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 4.5</td>
<td>The EHRRA must support a means of mapping between objects in information and inference models corresponding to a well-defined set of concepts in the foundation reference terminology (or concept) model. (1.4.1)</td>
<td>openEHR Data Types DV_CODED_TEXT and TERM_MAPPING types.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

### ISO Section 1.4.3  Language independence

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 4.6</td>
<td>The EHRRA must support the use of a comprehensive reference terminology that enables the recording/translation of multilingual terms. [This does not imply that a given EHR implementation must support more than one language].</td>
<td>Language is recorded in DV_TEXT instances; branching version control allows translations of complete Compositions to be recorded alongside the version in the original language.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>STR 4.7</td>
<td>The EHRRA must support the identification of information that has been translated from the language in which it was originally recorded. Such identification must describe the faithfulness or reliability of the translation. (1.4.3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ISO Section 1.4.4  Representation of text

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR 4.8</td>
<td>The original textual representation as entered by the clinician must be retained in the EHR when information is translated from one natural language to another or when terms are mapped from one coding/classification system to another.</td>
<td>Translation causes branched versioning in the openEHR EHR, ensuring that the original data are not obscured. Term mapping is supported via the TERM_MAPPING type.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
ISO Section 2  PROCESS

Preamble
The EHRRA must support clinical processes such as ordering, care planning, clinical guidelines, and decision support. It must also support processes associated directly with the record including the capture, retrieval, querying, presentation, and automatic processing of patient data. Good quality data is essential for good quality decision support and most other aspects of patient care, so uniform data capture methods and data definitions should be used whenever possible in EHR systems. The EHRRA should also support local clinical and workflow processes to ensure maximum usability and acceptability of EHR systems by clinicians and other users.

ISO Section 2.1  Clinical processes
ISO Section 2.1.1  Support for clinical processes

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.1</td>
<td>The EHRRA must support the recording of any type of clinical event, encounter, or episode relevant to the care of a patient (3.1)</td>
<td>The openEHR models are generic in nature, and do not directly model concepts such as “encounter” or “episode” - these are modelled by using archetypes, FOLDers, COMPOSITIONs and other elements of the architecture. All clinical events result in an “event” COMPOSITION, which contains relevant context in an attached EVENT_CONTEXT object.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>PRO 1.2</td>
<td>The EHRRA must support the creation, instantiation, and maintenance of clinical processes that support the activities of its users (3.3.5)</td>
<td>openEHR EHR RM: Archetypes can be used to define specific structures of the various kinds of ENTRY (ADMIN_ENTRY, OBSERVATION, EVALUATION, INSTRUCTION, and ACTION) and links between them describing causality or other relationships. As more events happen in a clinical process, changes to the states of INSTRUCTIONs/ACTIONs and the addition of links can be made, creating a growing picture of the real-world process as it unfolds in time. Integration with formal workflow systems is supported in INSTRUCTION and ACTION.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 1.3</td>
<td>The EHRRA must support the continuity of a clinical process, the ability to query the status of a process, modify an existing process, and verify that a process has been completed (3.3.5)</td>
<td>openEHR EHR RM: The INSTRUCTION type allow the state of fine-grained processes to be recorded using the DV_STATE data type. The state machines are defined in archetypes. When a process changes state, a new version of a COMPOSITION is made which records the state change. The status of coarse-grained processes such as care-plans is more likely to be recorded by clinicians as narrative.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 1.4</td>
<td>The EHRRA must be able to accommodate partial completion of a clinical process. (3.3.5)</td>
<td>openEHR EHR RM: The INSTRUCTION type includes a standard “Instruction State Machine” whose standard states can be mapped to workflow step names in particular clinical workflows in archetypes.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
## ISO Section 2.1.2  Problems/issues and health status

<table>
<thead>
<tr>
<th>ISO Req't</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.5</td>
<td>The EHRRA must support the recording and presentation of holistic health status, functional status, problems, conditions, environmental circumstances and issues (3.2.1)</td>
<td>openEHR EHR RM: all of these would be recorded in the appropriate “persistent” Compositions, according to appropriate archetypes.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>PRO 1.6</td>
<td>The EHRRA must support the recording and presentation of data in a problem-oriented structure including problem status, resolution plans and targets (problem-oriented here includes conditions and issues) (3.2.1)</td>
<td>openEHR EHR RM: particular types of persistent Composition are used to record problem list, issues, care plans etc; appropriate Section archetypes are used to support problem-oriented recording.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>PRO 1.7</td>
<td>The EHRRA must support a patient's lifetime, longitudinal record of health status and care interventions which can be viewed as a chronological health record. The patient EHR is at once (simultaneously): 1. retrospective: an historical view of health status and interventions (e.g., completed health service events/acts); 2. concurrent: a &quot;now&quot; view of health status and active interventions (e.g., health service events/acts now underway); and 3. prospective: a future view of planned interventions (e.g., health service events/acts scheduled or pending).</td>
<td>The openEHR models are designed to express the semantics of a longitudinal EHR, and are based on a number of core design principles [2]. The three views are supported by five subtypes of ENTRY, namely OBSERVATION, EVALUATION, INSTRUCTION, ACTION and ADMIN_ENTRY types which are based on some years of research and experience, including by GEHR, PEN&amp;PAD (U. Manchester), and by CEN and HL7.</td>
<td>Design: Full Val: CEN, GEHR, SNX.</td>
</tr>
</tbody>
</table>

## ISO Section 2.1.3  Clinical reasoning

<table>
<thead>
<tr>
<th>ISO Req't</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.8</td>
<td>The EHRRA must support the recording of the clinical reasoning including automated processes for all diagnoses, conclusions, and actions regarding the care of a patient (3.2.2)</td>
<td>openEHR EHR RM: Archetypes are used to define particular diagnoses and care plan information structures. Use of computerised clinical guidelines is supported via the CARE_ENTRY.guideline_id attribute.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>ISO Req’t</td>
<td>Description</td>
<td>openEHR artifact</td>
<td>Conformance</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PRO 1.9</td>
<td>The EHRRA must support the automatic presentation of warnings, alerts and reminders such as patient infective status, allergies and other therapeutic precautions, outstanding interventions, and urgent results (3.2.1)</td>
<td>openEHR EHR RM: Important items such as allergies, problems etc are most likely to be stored in a small number of often-accessed “persistent” Compositions in the EHR, each of which is identified by purpose, e.g. “therapeutic precautions”, “problem list” etc. In addition, any element of an openEHR EHR can be accessed via a URI-style path, allowing individual identification of important items. The actual detection and actioning of warnings and alerts is up to the calling application, such as a decision support system.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 1.10</td>
<td>The EHRRA must support systematic population-based recalls and reminders including public and population health programs such as immunisation and epidemiological surveillance (3.3.5)</td>
<td>openEHR EHR RM: The INSTRUCTION type in the EHR RM has been specifically designed to support automated recall management, including modelling of recall types with individual archetype-defined state machines. Paths to all recalls in an EHR would be added to a persistent Composition when they are first defined. This enables triggers to be created for each recall, based on its state machine and state data.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 1.11</td>
<td>The EHRRA must be able to support guidelines, protocols, and decision support systems (3.3.5)</td>
<td>The INSTRUCTION type in the EHR RM have been specifically designed to support automated guideline interaction. In particular:  - paths to items in the EHR needed by guidelines can be stored in the CARE_ENTRY . guideline_id object for the guideline;  - execution state of a guideline can be stored in the INSTRUCTION . state object for the guideline. Further experience and testing is needed in this area to determine whether the architecture needs to provide other support for decision support and other automated processing.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>PRO 1.12</td>
<td>The EHRRA must enable semantic interoperability of clinical concepts to support decision support processing.</td>
<td>Archetypes can be shared between the EHR and decision support systems, enabling decision support to search for semantically meaningful concepts rather than just atomic data items.</td>
<td>Design: Qual</td>
</tr>
</tbody>
</table>
### ISO Section 2.1.5 Care Planning

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
</table>
| PRO 1.13  | The EHRRA must support care planning, including the management of process states (e.g. planned, ordered, scheduled, in progress, on hold, pending, completed, amended, verified, cancelled), within the care planning process (3.2.4) | *openEHR EHR RM:* The EHR RM EVALUATION ENTRY subtype allows plans to be expressed, while the INSTRUCTION ENTRY type enables specific actions to be prescribed. The information defining any such action is expressed in an appropriate archetype, including the state machine definition for the process state.  
*openEHR Data Types:* The DV_STATE data type directly implements the concept of a state machine, and is designed to be driven by state machines defined on a per-archetype basis. | Design: Full |

### ISO Section 2.1.6 Orders & service processes

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.14</td>
<td>The EHRRA must support the recording and tracking of clinical orders and requests such as prescriptions and other treatment orders, investigation requests, and referrals (3.3.6)</td>
<td><em>openEHR EHR RM:</em> orders and other requests are recorded using the INSTRUCTION ENTRY type. Prescriptions are actually documents containing medication order requests from a provider to a filler such as a pharmacy. There may be an arbitrary relationship between medications and prescriptions from a given clinical session, due to a) prescriptions required for different fillers (e.g. a special drug may only be available from a specialist pharmacy), and b) due to legislation, e.g. in Australia, there can be a maximum of 3 medications on a prescription; and c) not all proposed medications or therapies need to be included in a prescription. Prescriptions should therefore be managed separately, and are most likely to be represented in openEHR as separate Compositions, or just with prescription ids from a prescribing system.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 1.15</td>
<td>The EHRRA must support the linking of orders with the observations that arise as a result (e.g. the results of an investigation or administration of a medication with the order for these interventions).</td>
<td><em>openEHR EHR RM and Data Types:</em> The LINK data type is provided for exactly this purpose, and may be used to create a named link between any archetyped data entities, i.e. COMPOSITIONs, SECTIONS, ENTRYs. Such links may be used to create causal chains or “problem threads” through the data.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
</tbody>
</table>
### ISO Section 2.1.7  Integrated care

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.16</td>
<td>The EHRRA must support integrated patient care including continuing collaborative multi-disciplinary care and case management across different healthcare sectors and settings (e.g. primary care, acute hospitals, allied health, home-based care) (3.2.3)</td>
<td>The EHR is actually agnostic about who records information in it and uses it. Access and care across different sectors are possible within the one EHR, since the architecture is generic, and does not correspond to any particular model or subdomain of care. However, it is up to EHR systems to actually enable access across different sectors, people etc.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
</tbody>
</table>

### ISO Section 2.1.8  Quality assurance

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 1.17</td>
<td>The EHRRA must support the recording and querying of data to enable the measurement of operational and clinical performance, to ensure compliance with standards of care, to ensure quality process and to measure outcomes.</td>
<td>There are no specific features of the reference models for supporting this. Any such data would be modelled using archetypes, and queried in the normal way.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>

### ISO Section 2.2  Record processes

#### ISO Section 2.2.1  Data capture

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 2.1</td>
<td>The EHRRA must support clear and consistent rules for entry, amendment, verification, transmittal, receipt, translation, and deletion of data. This requirement does not imply that it is necessary for a given implementation to allow deletion of EHR content. Local data retention rules will apply. (3.3.1)</td>
<td>openEHR EHR RM: all change to the EHR is governed by the semantics of version control built into the VERSIONED_OBJECT and VERSION classes. The fact of transmittal of EHR extracts to other users is not recorded in the EHR itself, since this is deemed to be the same as any other kind of non-modifying access. Where receipt of EHR extracts or other data such as messages causes changes to the EHR, the audit trailing indicates clearly where the data was acquired from.</td>
<td>Design: Part</td>
</tr>
<tr>
<td>ISO Req’t</td>
<td>Description</td>
<td>openEHR artifact</td>
<td>Conformance</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>PRO 2.2</td>
<td>The EHRRA must support the implementation of rules for data validation (3.3.1)</td>
<td>Archetypes: The openEHR archetypes are a key way of expressing constraints on data, including on type, value, structure and names, as well as on more esoteric things like allowed state transitions, fuzzy value mappings. These constraints provide a means of high-quality data validation.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 2.3</td>
<td>The EHRRA must support the ability to review information of all types recorded in the past, including via the use of query and filter facilities, during the data capture process (3.3.1)</td>
<td>openEHR EHR RM: The version control mechanism ensures that all previous states of Folders and Transactions in the EHR are preserved, and therefore any previous state of the EHR can be recreated.</td>
<td>Design: Full Val: CEN, GEHR</td>
</tr>
</tbody>
</table>

### ISO Section 2.2.2 Retrieval/query/Views of data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
</table>
| PRO 2.4   | The EHRRA must support selective retrieval and customized views of the same information for specific needs (e.g. decision support, data analysis) (3.3.2) | openEHR EHR RM: views can be created in various ways, including: 
  - using the VIEW ENTRY type, which enables the specification and optionally results of a query to be stored in the EHR; 
  - using FOLDERs to create coarse-grained views of Compositions in the record 
All references in views are defined as EHR Paths, using a standard URI-like textual referencing mechanism for any node or leaf in the EHR. | Design: Full Val: CEN, SNX |
### ISO Section 2.2.3  Presentation of data

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 2.5</td>
<td>The EHRRA must support the ability to display data marked as clinical summary without the need for manual searching (3.3.3)</td>
<td>Clinical summaries are likely to be stored in one or a small number of persistent Compositions based on a “clinical summary” archetype. Since persistent Compositions are likely to be stored in their own FOLDER, they are easy to find. Alternative approaches include: any Composition in which a clinical summary is included can have an entry in an index of archetype ids-&gt;Transactions, whereby Transactions containing any particular kind of information can be quickly found based on archetype id.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRO 2.6</td>
<td>The EHRRA must support the ability to convey the nature of devices on which information should by preference be presented where this may affect the clinical interpretation (eg viewing a colour image on a monochrome viewer, viewing a digital diagnostic image on a low resolution viewer) (3.3.3)</td>
<td>openEHR EHR RM: presumably this means that a clinical instruction about what kind of device to view the information on in order not to diminish its clinical utility should be included in the observation. Currently observation protocol can be recorded; should a “viewing protocol” also be included? This would seem to apply only for very particular circumstances.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>

### ISO Section 2.2.4  Scalability

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO 2.7</td>
<td>The EHRRA should not impede efficient processing of very large records or very large numbers of records.</td>
<td>openEHR EHR RM: Each EHR consists of VERSIONED_COMPOSITIONs whose most natural implementation is as separate entities in a database, ensuring performance does not diminish with size. Performance of systems containing large numbers of records is mostly a system issue, but is probably improved by the use of separate Compositions.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
Preamble
The principle underlying the requirements in this section is to enable data stored in EHRs to be transferred between different EHR systems and other clinical systems. Similarly, EHRs must be able to accept data transferred from different EHR systems and other clinical systems.

There are two distinct forms of transfer possible: messaging and record exchange. Messaging is necessary when data is transferred between systems which do not conform to the same EHR architecture standard. Messaging requires the use of agreed protocols such as HL7, UN/EDIFACT and DICOM. The format and methods of disseminating data must be standardised wherever possible.

Record exchange can occur where data is transferred between two EHR Systems that share a common architecture. Record exchange includes the movement or copying of all or part of an EHR.

ISO Section 3.1 Messaging

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM1.1</td>
<td>The EHRRA must support the export and import of data received using messaging protocols such as HL7, UN/EDIFACT and DICOM. (4)</td>
<td>Data from any other source can be incorporated into the record and represented in native openEHR form, as long as a mapping can be developed from the source form to openEHR. Transformations from CEN and HL7v2 and HL7v3 have been studied, and should be possible. Data in another format which cannot be converted can always be represented in encapsulated form.</td>
<td>Design: Fut</td>
</tr>
</tbody>
</table>
### ISO Section 3.2  Record exchange

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>COM 2.1</td>
<td>The EHRRA must allow for the exchange of a complete EHR or a part of an EHR (an extract) between EHRRA compliant systems. (4.4)</td>
<td>The semantics of EHR_EXTRACTs are formally defined, and allow any subset of the latest COMPOSITIONs of an EHR to be transmitted elsewhere. Transmitting a whole EHR means transmitting all its previous versions, and presumably has the semantics of “moving” rather than copying. This would be achieved by serialising the entire EHR according to the EHR &amp; VERSIONED_COMPOSITION classes (rather than the EHR_EXTRACT class) and transmitting it. However, the problem of transmitting demographic and terminology information also has to be addressed, and depends on what the intention is: is it about moving an entire EHR environment elsewhere, or just one patient?</td>
<td>Design: Qual Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>COM 2.2</td>
<td>The EHRRA must support serialisation of data for interoperability purposes (e.g. via XML, CORBA, SOAP, etc) (4.3)</td>
<td>The openEHR EHR RM supports any standard serialisation mechanism, such as XML, CORBA, .NET etc.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>COM 2.3</td>
<td>The EHRRA must define the semantics of merging data from an EHR extract with the EHR resident in the receiving system. (4.7)</td>
<td>openEHR EHR RM: the FEEDER_AUDIT (revision history) class and associated semantics are provided for just this purpose.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>COM 2.4</td>
<td>The EHRRA must provide an audit trail of exchange processes, including authentication, to enable identification of points of EHR extract transmittal and receipt. This needs to account for merging processes. (4.3)</td>
<td>openEHR EHR RM: the EHR_LOG object is designed to provide this facility. It has not been fully analysed.</td>
<td>Design: Fut</td>
</tr>
<tr>
<td>COM 2.5</td>
<td>The rules covering the exchange of an extract must be the same as those for exchanging the complete record. (4.4)</td>
<td>TBR 1: that depends - if the intention is to send a copy of the current state of an entire record for clinical/shared care purposes, this is true, and an EHR_EXTRACT can be used. If the intention is to move the whole record (e.g. to another jurisdiction, another information guardian), including previous versions, relevant demographic, terminological and access control data, the semantics will be different.</td>
<td>Design: TBD</td>
</tr>
<tr>
<td>COM 2.6</td>
<td>The EHRRA must enable semantic interoperability of clinical concepts between EHR systems to support automatic processing of data at the receiving system. (3.3.4)</td>
<td>The use of archetypes which are shared by both EHR systems in communication enables semantic interoperability between these parties.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
Preamble
The EHR must support the ethical and legal use of personal information, in accordance with established privacy principles and frameworks, which may be culturally or jurisdictionally specific. Key issues include control of access to the EHR to ensure personal health information can be kept confidential - ie used only for approved purposes and shared only among authorised people; and informed consent.
Key issues in relation to security include authentication, data integrity, confidentiality, non-repudiation and auditability.

ISO Section 4.1 Privacy and confidentiality

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRS1.1</td>
<td>The EHRRA must support the application of prevailing privacy and confidentiality rules. (5.2)</td>
<td>The LOCATABLE class, inherited into most other classes in the EHR RM, has an attribute archetype_details, of type ARCHETYPED. This latter class contains access_group identifiers, which can be used to refer to access control groups defined outside the EHR, e.g. in an authorisation service.</td>
<td>Design: Part</td>
</tr>
<tr>
<td>PRS1.2</td>
<td>The EHRRA must support the labelling of the whole and/or sections of the EHR as restricted to authorised users and/or purposes. This should include restrictions at the level of reading, writing, amendment, verification, and transmission/disclosure of data and records (5.2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRS1.3</td>
<td>The EHRRA must support privacy and confidentiality restrictions at the level of both data sets and discrete data attributes.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ISO Section 4.2 Consent

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRS2.1</td>
<td>The EHRRA must support recording of informed consent for the creation of a record. (5.3)</td>
<td>It appears that the consent data in all of these requirements can be adequately captured in a combination of INSTRUCTION and OBSERVATION_ENTRY types. It is not yet known whether all access control requirements would be covered. This requires further real-world experience.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>PRS2.2</td>
<td>The EHRRA must support obtaining, recording and tracking the status of informed consent to access the whole and/or sections of the EHR, for defined purposes. (5.3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRS2.3</td>
<td>The EHRRA must support recording of the purposes for which consent is obtained. (5.3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRS2.4</td>
<td>The EHRRA must support recording of the time frames attached to each consent. (5.3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### ISO Section 4.3  Access control

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRS3.1</td>
<td>The EHRRA must support measures to define, attach, modify and remove access rights to the whole and/or sections of the EHR. (5.1.1)</td>
<td><em>openEHR EHR RM:</em> Access control settings can be attached to any archetyped structure, including the EHR, a VERSIONED_COMPOSITION, a SECTION and an ENTRY. The access control settings are defined outside the EHR architecture. These may be changed at any time.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>PRS3.2</td>
<td>The EHRRA must support measures to define, attach, modify and remove access rights for classes of users of the EHR. (5.1.1)</td>
<td>The <em>openEHR</em> architecture does not define any particular model of access control, it just provides places to put access control settings at the lowest meaningful level of granularity, i.e. archetyped structures.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>PRS3.3</td>
<td>The EHRRA must support measures to enable and restrict access to the whole and/or sections of the EHR in accordance with prevailing consent and access rules. (5.1.1)</td>
<td>The details of how consent and access control are represented are not yet completed in the <em>openEHR</em> EHR RM.</td>
<td>Design: TBD</td>
</tr>
<tr>
<td>PRS3.4</td>
<td>The EHRRA must support measures to separately control authorities to add to and/or modify the EHR from authorities to access the EHR (5.1.1)</td>
<td><em>openEHR EHR RM:</em> This presumably means that access versus modification should be distinguished in the access control settings. Currently the facility exists to record any access rights, but not to force these two types to always be there.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>

### ISO Section 4.4  Data integrity

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRS4.1</td>
<td>The EHRRA must support measures to ensure the integrity of data stored in and transferred to and from EHRs (2.8.3)</td>
<td>The model for EHR Extracts does not include methods to guarantee integrity as such, since it is thought that these need to be applied to the serialised form of the data, which is dependent on the target technology, e.g. XML, CORBA etc. A model for general semantics of serialised form of extracts, including signing, may need to be described. Compositions may need a “digest”, i.e. a content-derived hash.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>
## ISO Section 4.5  
### Auditability of access

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRS5.1</td>
<td>The EHRRA must support recording of an audit trail of access to and modifications of data within the whole or sections of the EHR. (5.5)</td>
<td><em>openEHR EHR RM:</em> Modifications are audit trailed in the <code>AUDIT_DETAILS</code> of each <code>COMPOSITION</code>.</td>
<td>Design: Full/Fut</td>
</tr>
<tr>
<td>PRS5.2</td>
<td>The EHRRA must support recording of the nature of each access and/or transaction. (5.5)</td>
<td><em>openEHR EHR RM:</em> All modifications are audit-trailed in the <code>AUDIT_DETAILS</code> object of a <code>COMPOSITION</code>.</td>
<td>Design: Full/Fut</td>
</tr>
<tr>
<td>PRS5.3</td>
<td>The EHRRA must support audit capability sufficient to track accountability for each step or task in the clinical or operational processes recorded in the record. (5.5)</td>
<td><em>openEHR EHR RM:</em> Each distinct clinical observation, evaluation or analysis, and action recorded is represented using an <code>ENTRY</code>, which ensures the relevant context information for each such action is recorded.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
Preamble
Requirements for the medico-legal aspects of the EHRRA are essential if EHRs are to be trusted by both consumers and clinicians and accepted in courts of law as evidence of care provided, compliance with legislation, and the competence of clinicians. Many of the medico-legal requirements are related to and have implications for both privacy and security of the EHR but are nevertheless a distinct category.
For medico-legal purposes it is essential that every addition, amendment or alteration to the EHR be permanently recorded and preserved for an indefinite period. To maintain its originality, information must not be subject to later alteration or erasure. It is also essential that every actor be unambiguously identified and inextricably linked to the information for which they attest.
Legal requirements will vary widely among jurisdictions. In recognising these variances the EHR must not attempt to impose legal obligations of one society upon another. The EHRRA should ensure that the EHR can be a legally acceptable document in the jurisdiction in which it is created.

ISO Section 5.1 Support for legal requirements

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL1.1</td>
<td>The EHRRA must support measures to ensure an accurate reflection of the chronology of clinical events and information availability in the EHR (6.3)</td>
<td>openEHR EHR RM: the model specifically distinguishes between date/times of clinical events and acts, and interactions with the EHR system, ensuring that the chronology of events in the real world is clear, as well as the chronology of changes to the EHR.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>MEL1.2</td>
<td>The EHRRA must enable the viewing of an accurate representation of the EHR at any particular date and time since its creation (6.4)</td>
<td>openEHR EHR RM: the model explicitly includes versioning semantics in the VERSIONED_COMPOSITION and related classes.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

ISO Section 5.2 Actors

ISO Section 5.2.1 Subject of healthcare

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.1</td>
<td>The EHRRA must cater for the subject of care of the EHR to be one or more persons (6.1.1)</td>
<td>Currently any situation in which health information for a subject other than the subject of the record, e.g. a family member, a donated organ, or a foetus is well-understood and catered for via the subject attribute in ENTRY. Requirements for families, tribes, or other social groupings have not been widely described or analysed, but in any case can be accommodated by the same mechanism, limited only by what demographic entities can be represented.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
### ISO Section 5.2.2  Patient identification

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.2</td>
<td>The EHRRA must cater for the recording of appropriate patient identification attributes and clinically relevant patient attributes such as date of birth, sex, ethnicity etc (6.1.2)</td>
<td>openEHR Demographic RM: all PERSONs, include patients can have any number of identifiers and other data recorded for them.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
</tbody>
</table>

### ISO Section 5.2.3  User Identification

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.3</td>
<td>The EHRRA must ensure that users who attest and commit any particular information to the record are uniquely and reliably identified (6.1.3)</td>
<td>The EHR guarantees that distinct identifiers which occur in the EHR correspond to distinct demographic entities, i.e. that identifiers are not re-used. The ATTES-TATION type explicitly indicates the committer of the attestation.</td>
<td>Design: Full Val: GEHR, SNX</td>
</tr>
<tr>
<td>MEL2.4</td>
<td>The EHRRA must support the on-going ability to identify users, even if they change their name, profession, sex, or address. (6.1.3)</td>
<td>openEHR EHR RM: identifiers in the EHR referring to demographic entities managed in a demographic service always refer to the current information in that service.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

### ISO Section 5.2.4  Healthcare parties

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.5</td>
<td>The EHRRA must support measures to ensure that all clinical parties referred to in the EHR are uniquely identified (6.1.4)</td>
<td>openEHR EHR RM: Subtypes of the OBJECT_ID class are used to refer uniquely to demographic entities, including all clinical parties.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>MEL2.6</td>
<td>The EHRRA must support the recording of the clinical roles of any parties with respect to any clinical activity recorded. (6.1.4)</td>
<td>openEHR EHR RM: Currently roles and other demographic details are defined in the demographic model and its archetypes. Roles and relationships are included in the EHR architecture in the form of the PARTICIPATION and various PARTY_PROXY types.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
### ISO Section 5.2.5  Author responsibility

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.7</td>
<td>The EHRRA must support measures which ensure that every record entry is dated, its author identified. (6.1.6)</td>
<td>openEHR EHR RM: all additions to the record are in the form of COMPOSITIONs, which indicate the author’s identity, date, and time of addition.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>MEL2.8</td>
<td>The EHRRA must support measures to ensure that there is an absolute requirement that each contribution to the record is attributed to a responsible healthcare party, whether in the role of author or not. (6.1.5)</td>
<td>openEHR EHR RM: each COMPOSITION in the record identifies, via its sub-objects: • participations, e.g. including HCA legally responsible (EVENT_CONTEXT) • committer (AUDIT_DETAILS) • information_provider (ENTRY)</td>
<td>Design: Full Val: GEHR</td>
</tr>
</tbody>
</table>

### ISO Section 5.2.6  Attestation/Authorization of entries

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL2.9</td>
<td>The EHRRA must support measures which ensure that every contribution to the record must be attested by a responsible person. (6.1.6)</td>
<td>openEHR EHR RM: every COMPOSITION includes an AUDIT_DETAILS object which contains the mandatory attribute committer:PARTY_REF. An attestation form of the AUDIT_DETAILS object can be used if desired, allowing digital signing.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
<tr>
<td>MEL2.10</td>
<td>The EHRRA must support measures which ensure that amendments are attributed to a responsible person and the date and time and the reason for the amendment are recorded. (6.6)</td>
<td>openEHR EHR RM: amendments and new information are done by the same mechanism, namely COMPOSITIONs, which always include these details.</td>
<td>Design: Full Val: CEN, GEHR, SNX</td>
</tr>
</tbody>
</table>
### ISO Section 5.3  Clinical competence/governance

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL3.1</td>
<td>The EHRRA must support the demonstration of clinical competence and accountability of clinicians (6.2)</td>
<td>The openEHR approach ensures that any addition that clinicians make to the record identifies the author, the context of care and of recording, and allows reasons for doing things to be recorded (e.g. identifying guidelines etc). Links used to represent causal and other relationships enable chains of events to be followed back in time, ensuring that, as long as all actions are recorded in the record, the record will support the clinicians’ claims to have performed those actions. The ability to recreate any prior state of the record guarantees that any clinician’s claim about what information was available can be supported.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

### ISO Section 5.4  Faithfulness

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL4.1</td>
<td>The EHRRA must ensure that information intended to supersede that already recorded and attested must be separately collected and attested as a new transaction version. (6.5.1)</td>
<td>openEHR EHR RM: this is exactly the way the openEHR COMPOSITION concept works.</td>
<td>Design: Full Val: GEHR</td>
</tr>
<tr>
<td>MEL4.2</td>
<td>The EHRRA must ensure that the exact state of the record can be re-created for any given point of time since the original creation of the EHR.</td>
<td>openEHR EHR RM: this is enabled by the versioning mechanism.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>

### ISO Section 5.5  Preservation of context

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL5.1</td>
<td>Where coded terms in the EHR have been mapped to another coded terminology, the EHRRA must provide a means of indicating the faithfulness of the translation (6.5.2).</td>
<td>The openEHR Data Types RM: mappings between any text item (coded or not) and a coded term are explicitly modelled by the TERM_MAPPING class. The match attribute indicates the closeness of the match (broader, narrower, equivalent).</td>
<td>Design: Full</td>
</tr>
<tr>
<td>MEL5.2</td>
<td>The EHRRA must maintain the original context of all elements of the record irrespective of the potential separate distribution of elements (6.5.2)</td>
<td>The openEHR EHR model is based on a theory of context [2] which ensures that this exact requirement is always met.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
## ISO Section 5.6 Permanence

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL6.1</td>
<td>The EHRRA must ensure that attested information shall be stored in a protected mode, disallowing any changes or deletions. (6.6)</td>
<td>openEHR EHR RM: The VERSIONED_COMPOSITION class performs this function.</td>
<td>Design: Full Val: GEHR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL6.2</td>
<td>The EHRRA will ensure that amendments are attributed to a clinician and the date and time, and the reason for the amendment are recorded. (6.6)</td>
<td>openEHR EHR RM: The AUDIT_DETAILS class records the clinician authorising committal, and date/time of committal, among other things.</td>
<td>Design: Full Val: GEHR</td>
</tr>
</tbody>
</table>

## ISO Section 5.7 Version control

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL7.1</td>
<td>The EHRRA must incorporate a method of version control that supports information at the level at which it was attested (6.8)</td>
<td>openEHR EHR RM: The VERSIONED_COMPOSITION class performs this function.</td>
<td>Design: Full Val: GEHR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEL7.2</td>
<td>The EHRRA must support measures to discern modification or updating of the record using version control (6.8)</td>
<td>openEHR EHR RM: The VERSIONED_COMPOSITION class performs this function.</td>
<td>Design: Full Val: GEHR</td>
</tr>
</tbody>
</table>
ISO Section 6 ETHICAL

Preamble
The ethical and moral justification for the creation, storage and processing of health records derives from the fact that they are instrumental for the protection of health. The foundations of the relationship between a clinician and a patient are the delivery of clinical care to the highest standard and the respect for patient autonomy. This inevitably leads to the conclusion that the right to informed consent and the right to confidentiality are also ethical/moral principles of the highest importance.

ISO Section 6.1 Support for ethical justification

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETH1.1</td>
<td>The EHRRA must be able to record ethical approval for secondary uses of patient information held in the EHR (8)</td>
<td>No specific facility has been included for this purpose. It is expected that further research and development experience is required before this requirement can be analysed well enough to be implemented.</td>
<td>Design: TBD</td>
</tr>
</tbody>
</table>
ISO Section 7  CONSUMER/ CULTURAL

ISO Section 7.1  Consumer issues

Preamble

Benefits of EHRs for consumers
EHRs have the potential to significantly improve quality of care and health outcomes for consumers, primarily through availability to clinicians of accurate, current information about a consumer's healthcare history. Improved access to information for both consumers and clinicians has the potential to improve communication between consumers and clinicians, resulting in more meaningful consumer participation in the healthcare process. Having access to such information is empowering, enabling people to interact as informed consumers and make sensible choices within the healthcare system.

Accommodating the needs and interest of consumers raises issues of privacy, security, confidentiality and access.

Consumer aspects of privacy, security and confidentiality
Consumers of healthcare services must be secure in the knowledge that the information they share with their clinician is treated with respect for their privacy and kept secure and confidential. Otherwise, they will be unwilling to seek appropriate care or to provide accurate and complete information. This will not only compromise their own healthcare, but will also confound programmes of clinical and health services research, health professional education and public health promotion.

Consumers' point of view
EHRs will not only be accessible to consumers but also incorporate their views and comments resulting from self-monitoring of illness, dietary notes, notes on self-monitoring of sport and exercise performance, behavioural activities and moods, etc. Consumers may also use EHRs to seek advice about improving their health or ask questions about the management of their care. A consumer's point of view is important, supporting consumer involvement and promoting communication between consumers and clinicians.

Cultural issues
Cultural issues are an essential category of information to be recognised and accommodated in the requirements for EHRs. Many cultures do not support the idea of sharing patient information. Others share information and decision making on health matters at the level of the extended family or larger group. Some components of clinical competence are closely related to the role of clinicians in the societies in which they practice. The EHRRA must not impose the clinical practice of one society on the clinical practice of another, although it should promote ways of learning about different styles of clinical practice.

EHR development, therefore, needs to focus on community issues involving culture and consent, expectations, language, religious beliefs, individual identification and all these will determine the subsequent healthcare model.

ISO Section 7.1.1  Support for consumer issues

<table>
<thead>
<tr>
<th>ISO Req't</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>COC.1.1</td>
<td>The EHRRA must support the production of a consumer oriented view. (9.1)</td>
<td>openEHR EHR RM: Since all contributions to the record are marked with the author’s identification, a view of patient-added data are easily possible. If simplification of data &amp; presentation is required, either software applications or particular consumer-oriented archetypes would have to be developed.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>ISO Req't</td>
<td>Description</td>
<td>openEHR artifact</td>
<td>Conformance</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>COC1.2</td>
<td>The EHRRA must support consumers' right of access to all EHR information subject to jurisdictional constraints. (9.1)</td>
<td>The openEHR models do not predetermine any particular model of access; they support whichever access control model is required to be used in a given usage scope.</td>
<td>Design: Qual</td>
</tr>
<tr>
<td>COC1.4</td>
<td>The EHRRA must support consumers being able to incorporate self-care information, their point of view on personal healthcare issues, levels of satisfaction, expectations and comments they wish to record in EHRs. (9.1)</td>
<td>openEHR EHR RM: Self-care information is added in the same way as any information is added by a clinician. Personal comments, expectations etc can be added according to consumer-oriented archetypes which are developed for this purpose.</td>
<td>Design: Qual</td>
</tr>
</tbody>
</table>

### ISO Section 7.2 Cultural issues

#### ISO Section 7.2.1 Support for cultural issues

<table>
<thead>
<tr>
<th>ISO Req't</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>COC2.1</td>
<td>The EHRRA must support interoperability in a way that is truly global, yet respects local customs and culture. It follows that the process must be both simple and amenable to customisation in different jurisdictions. (9.2)</td>
<td>General support for interoperability occurs with the approach of using a generic model which does not describe any specific clinical or medical concepts. openEHR Data Types: Data interoperability is supported as follows: • all coded terms include terminology identifier • all text types use UNICODE</td>
<td>Design: Full</td>
</tr>
<tr>
<td></td>
<td></td>
<td>openEHR EHR RM: Other aspects of interoperability: • the structure of the record is essentially container/headings/structured data, and data may be as minimally or maximally structured as desired. openEHR Archetype systems: The archetype system provides the most powerful basis for semantic interoperability, while allowing local definition and customisation of archetypes according to required medical and social cultures.</td>
<td></td>
</tr>
</tbody>
</table>
Preamble
To enable the creation and maintenance of life-time longitudinal electronic health records, it is necessary to ensure that both EHRs and EHR software are "future proof". Technology will continue to change rapidly. This means that the EHRRA must be effectively technology independent. The EHR architecture must therefore be able to accommodate new forms of clinical knowledge (e.g. genomics and proteomics) which may include not only new clinical content but also completely new types of data. On the other hand, legacy systems will persist long into the future and it is therefore necessary that a standard-compliant EHRRA must be able to support legacy data.

ISO Section 8.1 Support for EHR architecture and EHR system evolution

<table>
<thead>
<tr>
<th>ISO Req’t</th>
<th>Description</th>
<th>openEHR artifact</th>
<th>Conformance</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVO1.1</td>
<td>Backwards compatibility of EHR software: Any implementation of the EHRRA must be able to process EHRs created under older versions of the EHRRA (10.1.1)</td>
<td>This requirement is satisfied by the use of appropriate rules for what constitutes a new version of the openEHR architecture. New software versions are not allowed to invalidate previous data model elements, only add new ones, guaranteeing backwards compatibility of software.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>EVO1.2</td>
<td>Backwards compatibility of the EHR: Software built on a previous version of the EHRRA must be capable of processing EHRs created under a newer version of the EHRRA (10.1.1)</td>
<td>This requirement is satisfied by the use of appropriate rules for what constitutes a new version of the openEHR architecture. New versions are not allowed to invalidate previous data model elements, only add new ones, guaranteeing forward compatibility of data.</td>
<td>Design: Full</td>
</tr>
<tr>
<td>EVO1.3</td>
<td>The EHRRA must be able to accommodate the recording of information due to new forms of clinical knowledge, new clinical disciplines, and new clinical practices and processes (10.1.1)</td>
<td>The openEHR two-level modelling approach (bottom level = reference models; second level = domain concept models, or archetypes) is designed precisely to satisfy this requirement in a formal, systematic way.</td>
<td>Design: Full</td>
</tr>
</tbody>
</table>
3  Summary of Conformance Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBD</td>
<td>14</td>
</tr>
<tr>
<td>Full</td>
<td>18</td>
</tr>
<tr>
<td>Full</td>
<td>21</td>
</tr>
<tr>
<td>Full</td>
<td>21</td>
</tr>
<tr>
<td>TBD</td>
<td>26</td>
</tr>
<tr>
<td>Part</td>
<td>26</td>
</tr>
<tr>
<td>TBD</td>
<td>26</td>
</tr>
<tr>
<td>TBD</td>
<td>28</td>
</tr>
<tr>
<td>Fut</td>
<td>28</td>
</tr>
<tr>
<td>Fut</td>
<td>29</td>
</tr>
<tr>
<td>TBD</td>
<td>30</td>
</tr>
<tr>
<td>Part</td>
<td>30</td>
</tr>
<tr>
<td>TBD</td>
<td>30</td>
</tr>
<tr>
<td>TBD</td>
<td>31</td>
</tr>
<tr>
<td>TBD</td>
<td>32</td>
</tr>
<tr>
<td>TBD</td>
<td>32</td>
</tr>
<tr>
<td>Full</td>
<td>32</td>
</tr>
<tr>
<td>Full</td>
<td>34</td>
</tr>
<tr>
<td>Full</td>
<td>35</td>
</tr>
<tr>
<td>Full</td>
<td>36</td>
</tr>
<tr>
<td>TBD</td>
<td>39</td>
</tr>
</tbody>
</table>
4 ISO Requirements Requiring Review

TBR 1: ................................................................. 30
A References

Publications


3. Fowler M, Scott K. *UML Distilled (2nd Ed.)*. Addison Wesley Longman 2000


Resources


10. GEHR - Good European Health Record. Available at [http://www.chime.ucl.ac.uk/](http://www.chime.ucl.ac.uk/).
END OF DOCUMENT
openEHR Architecture

Architecture Overview

Editors: {T Beale, S Heard}¹, {D Kalra, D Lloyd}²

Revision: 1.0.2

Pages: 45

1. Ocean Informatics Australia
2. Centre for Health Informatics and Multi-professional Education, University College London

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
# Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.2</td>
<td>CR-000203: Release 1.0 explanatory text improvements. Improved path explanation. Slight re-ordering of main headings.</td>
<td>T Beale</td>
<td>06 Mar 2006</td>
</tr>
<tr>
<td>1.0.1</td>
<td>CR-000200: Correct package names in RM diagram. Added configuration management and versioning material from Common IM. Added section on ontological landscape. Added section on aims. Added section on systems architectures.</td>
<td>D Lloyd, T Beale</td>
<td>23 Feb 2006</td>
</tr>
</tbody>
</table>

## RELEASE 1.0

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
</table>
Acknowledgements

The work reported in this paper has been funded by the University College, London; Ocean Informatics, Australia.

CORBA is a trademark of the Object Management Group

.Net is a trademark of Microsoft Corporation
10.3 EHR URIs ............................................................................................................ 38
10.3.1 Locating Top-Level Structures ..................................................................... 38

11 Relationship to Standards ................................................................................. 40

12 Implementation Technology Specifications ................................................. 42
12.1 Overview .......................................................................................................... 42

A References ............................................................................................................ 43
1 Introduction

1.1 Purpose

This document provides an overview of the openEHR architecture in terms of a model overview, key global semantics, relationship to published standards, and finally the approach to building Implementation Technology Specifications (ITSs). Semantics specific to each information, archetype and service model are described in the relevant model.

The intended audience includes:

- Standards bodies producing health informatics standards
- Software development groups using openEHR
- Academic groups using openEHR
- The open source healthcare community

1.2 Related Documents

Prerequisite documents for reading this document include:

- The openEHR Roadmap document
- The openEHR Modelling Guide

Other documents describing related models, include:

- The openEHR Information Model documents
- The openEHR Archetype Model documents

1.3 Status

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

This document is available at http://svn.openeehr.org/specification/TAGS/Release-1.0/publishing/architecture/overview.pdf.

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

New versions are announced on openehr-announce@openehr.org.

Blue text indicates sections under active development.

1.4 Peer review

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

To Be Continued:   more work required

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

This document provides an overview of the openEHR architecture. It commences with a description of the specification project, followed by an overview of the reference model structure and packages. Key global semantics including archetyping, identification, version and paths are then described. The relationship to published standards is indicated, and finally, the approach to building Implementation Technology Specifications (ITSs) is outlined.

2.1 The openEHR Specification Project

FIGURE 1 illustrates the openEHR Specification Project. The project consists of requirements, architectural specifications, implementation technology specifications (ITSs), and conformance specifications. The focus of this document is the architectural and implementation technology specifications (ITSs).

![FIGURE 1 openEHR Specification project](image)

The architecture specifications consist of the Reference Model (RM), the Service Model (SM) and Archetype Model (AM). The first two correspond to the ISO RM/ODP information and computational viewpoints respectively.

All of the architecture specifications published by openEHR are defined as a set of abstract models, using the UML notation and formal textual class specifications. These models remain the primary references for all semantics, regardless of what is done in any implementation domain. The openEHR Modelling Guide describes the semantics of the models. The presentation style of these abstract specifications is deliberately intended to be clear, and semantically close to the ideas being communicated. Conversely, the abstract specifications do not follow idioms or limitations of particular programming languages, schema languages or other formalisms. All such expressions are treated as ITSs, for which explicit mappings generally have to be developed and described (since almost no formalism natively implements complete UML semantics).

There are numerous implementation technologies, ranging from programming languages, serial formalisms such as XML, to database and distributed object interfaces. Each of these has its own limits and strengths. The approach to implementing any of the openEHR abstract models in a given implementation technology is to firstly define an “implementation technology specification” (ITS) for the particular technology, then to use it to formally map the abstract models into expressions in that technology.
3 Aims

Types of System
This section provides a brief overview of the aims of the openEHR specifications, as an aid to understanding the remainder of the document. The architecture of openEHR is designed to support the construction of a number of types of system. One of the most important could be characterised as a distributed, patient-centred, life-long, shared care health record, illustrated in FIGURE 2.

In this form, the openEHR services are added to the existing IT infrastructure provide a shared, secure health record for patients that are seen by any number of health providers in their community context. openEHR-enabled systems can also be used to provide EMR/EPR functionality at provider locations. Overall, a number of important categories of system can be implemented using openEHR including the following:

- shared-care community or regional health service EHRs;
- summary EHRs at a national, state, province or similar level;
- small desktop GP systems;
- hospital EMRs;
- consolidated and summary EHRs in federation environments;
- legacy data purification and validation gateways;
- web-based secure EHR systems for mobile patients.

Requirements
The openEHR architecture embodies 15 years of research from numerous projects and standards from around the world. It has been designed based on requirements captured over many years. Among the global requirements of EHRs and EHR systems supported by openEHR are the following:

- a life-long EHR;
• prioritises the patient / clinician interaction;
• medico-legal faithfulness, traceability, audit-trailing;
• technology & data format independent;
• facilitate sharing of EHRs via interoperability at data and knowledge levels;
• suitable for both primary & acute care;
• integrates with any/multiple terminologies;
• supports all natural languages, as well as translations between languages in the record;
• support for patient privacy, including anonymous EHRs;
• support for clinical data structures: lists, tables, time-series, including point and interval events;
• support for all aspects of pathology data, including normal ranges, alternative systems of units etc;
• highly maintainable and flexible software;
• compatibility with CEN 13606, Corbamed, and messaging systems;
• support semi-automated and automated distributed workflows;
• supports secondary uses: education, research, population medicine.

One comprehensive statement of EHR requirements covering many of the above is the ISO Technical Report 18308\(^1\) for which an openEHR profile has been created\(^2\). The requirements summarised above are described in more detail in the openEHR EHR Information Model document.

---

4 Design Principles

The openEHR approach to modelling information, services and domain knowledge is based on a number of design principles, described below. All of these principles lead to a separation of the models of the openEHR architecture, and consequently, a high level of componentisation. This leads to better maintainability, extensibility, and flexible deployment.

4.1 Ontological Separation

The most basic kind of separation in any system of models is ontological, i.e. in the semantic dimension. All models carry some kind of semantic meaning, but not all semantics are the same, or even of the same category. For example, some part of the SNOMED-CT terminology will describe types of bacterial infection, sites in the body, and symptoms. An information model might specify a logical type Quantity. A content model might define the model of information collected in an ante-natal examination by a physician. These types of information are qualitatively different, and need to be developed and maintained separately within the overall model eco-system. FIGURE 3 illustrates these distinctions, and indicates what parts are built directly into software and databases.

This figure shows a primary separation between “ontologies of information” i.e. models of information content, from “ontologies of reality” i.e. descriptions and classifications of real phenomena. These two categories have to be separated because the authors, the representation and the purposes are completely different. In health informatics, this separation already exists by and large, due to the development of terminologies and classifications.

A secondary ontological separation within the “information” side is shown between information models and domain content models. This separation is not generally well understood, and historically, the entirety of informational semantics has been hard-wired into the software and databases, leading to unmaintainable systems. One of the key paradigms on which openEHR is based is known as “two-level” modelling, described in [2]. Under the two-level approach, a stable reference information...
model constitutes the first level of modelling, while formal definitions of clinical content in the form of archetypes and templates constitute the second. Only the first level is implemented in software, significantly reducing the dependency of deployed systems and data on variable content definitions. The only other parts of the model universe implemented in software are highly stable languages/models of representation (shown at the bottom of the figure). As a consequence, systems have the possibility of being far smaller and more maintainable. They are also inherently self-adapting, since they are built to consume archetypes and templates as they are developed into the future.

Archetypes and templates also act as a well-defined semantic gateway to terminologies, classifications and computerised clinical guidelines. The alternative in the past has been to try to make systems function solely with a combination of hard-wired software and terminology. This approach is flawed, since terminologies don’t contain definitions of domain content (e.g. “microbiology result”), but rather facts about the real world (e.g. kinds of microbes and the effects of infection in humans).

By clearly separating the three elements - information models, domain content models, and terminologies - the openEHR architecture enables each to have a well-defined, limited scope and clear interfaces. This limits the dependence of each on the other, leading to more maintainable and adaptable systems. The details of how archetypes and templates work in openEHR are described in Archetyping on page 31.

4.2 Separation of Responsibilities

A second key design paradigm used in openEHR is that of separation of responsibilities within the computing environment. Complex domains are only tractable if the functionality is first partitioned into broad areas of interest, i.e. into a “system of systems” [5]. This principle has been understood in computer science for a long time under the rubrics “low coupling”, “encapsulation” and “componentisation”, and has resulted in highly successful frameworks and standards, including the OMG’s CORBA specifications and the ISO Reference Model for Open Distributed Processing (RM-ODP) [4]. Each area of functionality forms a focal point for a set of models formally describing that area, which, taken together usually correspond to a distinct information system or service.

FIGURE 4 illustrates a notional health information environment containing numerous services, each denoted by a bubble. Typical connections are indicated by lines, and bubbles closer to the centre correspond to services closer to the core needs of clinical care delivery, such as the EHR, terminology, demographics/identification and medical reference data. Of the services shown on the diagram, openEHR currently provides specifications only for the more central ones, including EHR and Demographics.

Since there are standards available for some aspects of many services, such as terminology, image formats, messages, EHR Extracts, service-based interoperability, and numerous standards for details such as date/time formats and string encoding, the openEHR specifications often act as a mechanism to create coherent structural definitions in the informational and computational viewpoints that integrate existing standards.

4.3 Separation of Viewpoints

The third computing paradigm used in openEHR is a natural consequence of the separation of responsibilities, namely the separation of viewpoints. When responsibilities are divided up among distinct components, it becomes necessary to define a) the information that each processes, and b) how they will communicate. These two aspects of models constitute the two central “viewpoints” of the ISO RM/ODP model [4], which are as follows:
FIGURE 4 A Health Information Environment
**Enterprise:** concerned with the business activities, i.e. purpose, scope and policies of the specified system.

**Information:** concerned with the semantics of information that needs to be stored and processed in the system.

**Computational:** concerned with the description of the system as a set of objects that interact at interfaces - enabling system distribution.

**Engineering:** concerned with the mechanisms supporting system distribution.

**Technological:** concerned with the detail of the components from which the distributed system is constructed.

The openEHR specifications accordingly include an information viewpoint - the openEHR Reference Model - and a computational viewpoint - the openEHR Service Model. The Engineering viewpoint corresponds to the Implementation Technology Specification (ITS) models of openEHR (see Implementation Technology Specifications on page 42), while the Technological viewpoint corresponds to the technologies and components used in an actual deployment. An important aspect of the division into viewpoints is that there is generally not a 1:1 relationship between model specifications in each viewpoint. For example, there might be a concept of “health mandate” (a CEN ENV13940 Continuity of Care concept) in the enterprise viewpoint. In the information viewpoint, this might have become a model containing many classes. In the computational viewpoint, the information structures defined in the information viewpoint are likely to recur in multiple services, and there may or may not be a “health mandate” service. The granularity of services defined in the computational viewpoint corresponds most strongly to divisions of function in an enterprise or region, while the granularity of components in the information viewpoints corresponds to the granularity of mental concepts in the problem space, the latter almost always being more fine-grained.
5 openEHR Package Structure

5.1 Overview

FIGURE 6 illustrates the overall package structure of the openEHR formal specifications. Three major packages are defined: rm, am and sm. All packages defining detailed models appear inside one of these outer packages, which may also be thought of as namespaces. They are conceptually defined within the org.openehr namespace, which is usually represented in UML as further packages. In some implementation technologies (e.g. java), the org.openehr namespace may actually be used within program texts.

FIGURE 5 Global Package Structure of openEHR

5.2 Reference Model (RM)

Within the any given namespace, each package defines a local context for definition of classes. FIGURE 6 illustrates the package structure in the RM namespace. An informal division into “scientific computing” and “health information” is shown. The packages in the latter group are generic, and are used by all openEHR models, in all the outer packages. Together, they provide identification, access to knowledge resources, data types and structures, versioning semantics, and support for archetyping. The packages in the former group define the semantics of enterprise level health information types, including the EHR and demographics.

Each outer package in FIGURE 6 corresponds to one openEHR specification document\(^1\), documenting an “information model” (IM). Where packages are nested, the inner packages cannot exist outside of their parent package. The package structure will normally be replicated in all ITS expressions, e.g. XML schema, programming languages like Java, C# and Eiffel, and interoperability definitions like IDL and .Net.

5.2.1 Package Overview

The following sub-sections provide a brief overview of the RM packages.

Support Information Model

This package describes the most basic concepts, required by all other packages, and is comprised of the Definitions, Identification, Terminology and Measurement packages. The semantics defined in

---

\(^1\) with the exception of the EHR and Composition packages, which are both described in the EHR Reference Model document; this may change in the future.
FIGURE 6  Structure of org.openehr.rm package
these packages allow all other models to use identifiers and to have access to knowledge services like terminology and other reference data. The support package includes the special package assumed_types, describing what basic types are assumed by openEHR in external type systems; this package is a guide for integrating openEHR models proper into the type systems of implementation technologies.

**Data Types Information Model**

A set of clearly defined data types underlies all other models, and provides a number of general and clinically specific types required for all kinds of health information. The following categories of datatypes are defined in the data types reference model.

- **Text**: plain text, coded text, paragraphs.
- **Quantities**: any ordered type including ordinal values (used for representing symbolic ordered values such as “+”, “+++”, “++++”), measured quantities with values and units, and so on.
- **Date/times**: date, time, date-time types, and partial date/time types.
- **Encapsulated data**: multimedia, parsable content.
- **Basic types**: boolean, state variable.

**Data Structures Information Model**

In many reference models, generic data structures are available for expressing content whose particular structure will be defined by archetypes. The generic structures are as follows.

- **Single**: single items, used to contain any single value, such as a height or weight.
- **List**: linear lists of named items, such as many pathology test results.
- **Table**: tabular data, including unlimited and limited length tables with named and ordered columns, and potentially named rows.
- **Tree**: tree-shaped data, which may be conceptually a list of lists, or other deep structure.
- **History**: time-series structures, where each time-point can be an entire data structure of any complexity, described by one of the above structure types. Point and interval samples are supported.

**Common Information Model**

Several semantic concepts are used in common by various models. The classes `LOCATABLE` and `ARCHETYPED` provide the link between information and archetype models. The classes `ATTESTATION` and `PARTICIPATION` are generic domain concepts that appear in various reference models. The last group of concepts consists of a formal model of change management which applies to any service that needs to be able to supply previous states of its information, in particular the demographic and EHR services.

**EHR Information Model**

The EHR IM defines the containment and context semantics of the concepts `EHR`, `COMPOSITION`, `SECTION`, and `ENTRY`. These classes are the major coarse-grained components of the EHR, and correspond directly to the classes of the same names in CEN EN13606:2005 and fairly closely to the “levels” of the same names in the HL7 Clinical Document Architecture (CDA) release 2.0.

**EHR Extract**

The EHR Extract IM defines how an EHR extract is built from `COMPOSITIONs`, demographic, and access control information from the EHR.
Demographics
The demographic model defines generic concepts of \texttt{PARTY}, \texttt{ROLE} and related details such as contact addresses. The archetype model defines the semantics of constraint on \texttt{PARTY}s, allowing archetypes for any type of person, organisation, role and role relationship to be described. This approach provides a flexible way of including the arbitrary demographic attributes allowed in the OMG HDTF PIDS standard.

Workflow
Workflow is the dynamic side of clinical care, and consists of models to describe the semantics of processes, such as recalls, as well as any care process resulting from execution of guidelines.

5.3 Archetype Model (AM)
The \texttt{openEHR am} package contains the models necessary to describe the semantics of archetypes and templates, and their use within \texttt{openEHR}. These include ADL, the Archetype Definition Language (expressed in the form of a syntax specification), the \texttt{archetype} and \texttt{template} packages, defining the object-oriented semantics of archetypes and templates, and the \texttt{openehr_profile} package, which defines a profile of the generic archetype model defined in the \texttt{archetype} package, for use in \texttt{openEHR} (and other health computing endeavours). The internal structure of the \texttt{am} package is shown in FIGURE 7.

![FIGURE 7 Structure of the org.openehr.am package](image)

5.4 Service Model (SM)
The \texttt{openEHR} service model includes definitions of basic services in the health information environment, centred around the EHR. It is illustrated in FIGURE 8. The set of services actually included will undoubtedly evolve over time, so this diagram should not be seen as definitive.

Virtual EHR API
The virtual EHR API defines the fine-grained interface to EHR data, at the level of Compositions and below. It allows an application to create new EHR information, and to request parts of an existing EHR and modify them. This API enables fine-grained archetype-mediated data manipulation. Changes to the EHR are committed via the EHR service.
**EHR Service Model**

The EHR service model defines the coarse-grained interface to electronic health record service. The level of granularity is openEHR Contributions and Compositions, i.e. a version-control / change-set interface. The finest object that can be requested or committed via the EHR service is a single Composition, or the EHR Directory structure.

Part of the model defines the semantics of server-side querying, i.e. queries which cause large amounts of data to be processed, generally returning small aggregated answers, such as averages, or sets of ids of patients matching a particular criterion.

![Structure of the org.openehr.sm package](FIGURE_8)

**Archetype Service Model**

The archetype service model defines the interface to online repositories of archetypes, and can be used both by GUI applications designed for human browsing as well as access by other software services such as the EHR.

**Terminology Interface Model**

The terminology interface service provides the means for all other services to access any terminology available in the health information environment, including basic classification vocabularies such as ICDx and ICPC, as well as more advanced ontology-based terminologies. Following the concept of division of responsibilities in a system-of-systems context, the terminology interface abstracts the different underlying architectures of each terminology, allowing other services in the environment to access terms in a standard way. The terminology service is thus the gateway to all ontology- and terminology-based knowledge services in the environment, which along with services for accessing guidelines, drug data and other “reference data” enables inferencing and decision support to be carried out in the environment.
6 Scope of Architecture

6.1 Correspondence to System Architectures

The previous section described the software package structure of the openEHR specifications. Here we describe how the package architecture can be applied to building real systems. The general architectural approach in any openEHR system can be considered as 5 layers (i.e. a “5-tier” architecture). The tiers are as follows.

1. **persistence**: data storage and retrieval.
2. **back-end services**: including EHR, demographics, terminology, archetypes, security, record location, and so on. In this layer, the separation of the different services is transparent, and each service has a coarse-grained service interface.
3. **virtual EHR**: this tier is the middleware, and consists of a coherent set of APIs to the various back-end services providing access to the relevant services, thereby allowing user access to the EHR; including EHR, demographics, security, terminology, and archetype services. It also contains an archetype- and template-enabled kernel, the component responsible for creating and processing archetype-enabled data. In this tier, the separation of back-end services is hidden, only the functionality is exposed. Other virtual clients are possible, consisting of APIs for other combinations of back-end services.
4. **application logic**: this tier consists of whatever logic is specific to an application, which might be a user application, or another service such as a query engine.
5. **presentation layer**: this layer consists of the graphical interface of the application, where applicable.

The same tiers can be used in large deployments, as shown in FIGURE 9, or simply as layers in single-machine applications.

![FIGURE 9 Basic Enterprise EHR System Architecture](image-url)
FIGURE 10 illustrates an approximate mapping of major parts of the openEHR software architecture to the 5-tier scheme. Clearly where parts of the architecture are used will depend on various implementation choices; the mapping shown is therefore not definitive. Nevertheless, the principal use of parts of the architecture is likely to be similar in most systems, as follows:

- RM and AM: mainly used to construct an archetype- and template-processing kernel;
- RM `common.change_control` package: provides the logic for versioning in versioned services such as the EHR and demographics;
- SM: various service model packages define the exposed interfaces of major services;
- SM `virtual_ehr` package defines the API of the virtual EHR component;
- archetypes: archetypes might be assumed directly in some applications, e.g. a specialist peri-natal package might be partly based on a family of archetypes for this specialisation;
- templates: both archetypes and templates will be used in the presentation layer of applications. Some will base the GUI code on them, while others will have either tool-generate code, or dynamically generate forms based on particular templates and archetypes.

In the future, an abstract persistence API and optimised persistence models (transformations of the existing RM models) are likely to be published by openEHR in order to help with the implementation of databases.

![FIGURE 10 Mapping of software architecture to systems](image)

### 6.2 Top-level Information Structures

The openEHR information models define various informational artifacts at varying levels of granularity. Fine-grained structures defined in the Support and Data types are used in the Data Structures and Common models; these are used in turn in the EHR, EHR Extract, Demographic and other “top-level” models. These latter models define the “top-level structures” of openEHR, i.e. content structures that can sensibly stand alone, and may be considered the equivalent of separate documents in a document-oriented system. In openEHR information systems, it is the top-level structures that are of direct interest to users. The major top-level structures include the following:

- **Composition** - the committal unit of the EHR (see type `COMPOSITION` in EHR IM);
- **EHR Status** - the status summary of the EHR (see type `EHR_STATUS` in EHR IM);
- **Folder hierarchy** - act as directory structures in EHR, Demographic services (see type `FOLDER` in Common IM);
- **Party** - various subtypes including Actor, Role, etc representing a demographic entity with identity and contact details (see type `PARTY` and subtypes in Demographic IM);
- **EHR Extract** - the transmission unit between EHR systems, containing a serialisation of EHR, demographic and other content (see type `EHR_EXTRACT` in EHR Extract IM).

All persistent openEHR EHR, demographic and related content is found within top-level information structures.
7 Versioning

7.1 Overview

Version control is an integral part of the openEHR architecture. An openEHR repository for EHR or demographic information is managed as a change-controlled collection of “version containers” (modelled by the VERSIONED_OBJECT<T> class in the common.change_control package), each containing the versions of a top-level content structure (such as a Composition or Party) as it changes over time. A version-controlled top-level content structure is visualised in FIGURE 11.

[FIGURE 11 Version-control structures]

Versioning of single top-level structures is a necessary, but not sufficient requirement for a repository that must provide coherence, traceability, indelibility, rollback, and support for forensic examination of past states of the data. Features supporting “change control” are also required. Under a disciplined change control scheme, changes are not made arbitrarily to single top-level structures, but to the repository itself. Changes take the form of change-sets, called “contributions”, that consist of new or changed versions of the controlled items in the repository. The key feature of a change-set is that it acts like a transaction, and takes the repository from one consistent state to another, whereas arbitrary combinations of changes to single controlled items could easily be inconsistent, and even dangerously wrong where clinical data are concerned.

These concepts are well-known in configuration management (CM), and are used as the basis for most software and other change management systems, including numerous free and commercial products available today. They are a central design feature of openEHR architecture. The following sections provide more details.

7.2 The Configuration Management Paradigm

The “configuration management” (CM) paradigm is well-known in software engineering, and has its own IEEE standard\(^1\). CM is about managed control of changes to a repository of items (formally called “configuration items” or CIs), and is relevant to any logical repository of distinct information items which changes in time. In health information systems, at least two types of information require such management: electronic health records, and demographic information. In most analyses in the past, the need for change management has been expressed in terms of specific requirements for audit trailing of changes, availability of previous states of the repository and so on. In openEHR, the aim is

---

to provide a formal, general-purpose model for change control, and show how it applies to health information.

### 7.2.1 Organisation of the Repository

The general organisation of a repository of complex information items such as a software repository, or the EHR consists of the following:

- a number of distinct information items, or *configuration items*, each of which is uniquely identified, and may have any amount of internal complexity;
- optionally, a directory system of some kind, in which the configurations items are organised;
- other environmental information which may be relevant to correctly interpreting the primary versioned items, e.g. versions of tools used to create them.

In a software or document repository, the CIs are files arranged in the directories of the file system; in an EHR based on *openEHR*, they are Compositions, the optional Folder structure, Parties in the demographic service and so on. Contributions are made to the repository by users. This general abstraction is visualised in FIGURE 12.

![General Structure of a Controlled Repository](image)

**FIGURE 12** General Structure of a Controlled Repository

### 7.2.2 Change Management

Change doesn’t occur to CIs in isolation, but to the repository as a whole. Possible types of change include:

- creation of a new CI;
- removal of a CI;
- modification of a CI;
- creation of, change to or deletion of part of the directory structure;
- moving of a CI to another location in the directory structure.

The goal of configuration management is to ensure the following:
• the repository is always in a valid state;
• any previous state of the repository can be reconstructed;
• all changes are audit-trailed.

7.3 Managing Change in Time

Properly managing changes to the repository requires two mechanisms. The first, version control, is used to manage versions of each CI, and of the directory structure if there is one. The second is the concept of the “change-set”, known as a contribution in openEHR. This is the set of changes to individual CIs (and the directory structure) made by a user as part of some logical change. For example, in a document repository, the logical change might be an update to a document that consists of multiple files (CIs). There is one contribution, consisting of changes to the document file CIs, to the repository. In the EHR, a contribution might consist of changes to more than one Composition, and possibly to the organising Folder structure.

A typical sequence of changes to a repository is illustrated below. FIGURE 13 shows the effect of four Contributions (indicated by blue ovals on the left hand side) to a repository containing a number of CIs (that the directory tree is not shown for the sake of simplicity). As each Contribution is made, the repository is changed in some way. The first brings into existing a new CI, and modifies three others (changes indicated by the ‘C’ triangles). The second contribution causes the creation of a new CI only. The third causes a creation as well as two changes, while the fourth causes only a change. (Changes to the folder structure are not shown here).

One nuance which should be pointed out is that in FIGURE 13, contributions are shown as if they are literally a set of deltas, i.e. exactly the changes which occur to the record. Thus, the first contribution is the set \{CI_w, C_{a1}, C_{c1}, C_{d1}\} and so on. Whether this is literally true depends on the construction of applications. In some situations, some CIs may be updated by the user viewing the current list and entering just the changes - the situation shown in FIGURE 13; in others, the system may provide the current state of these CIs for editing by the user, and submit the updated versions, as shown in FIG-
URE 14. Some applications may do both, depending on which CI is being updated. The internal versioning implementation may or may not generate deltas as a way of efficient storage.

For the purposes of openEHR, a contribution is considered as being the logical set of CIs changed or created at one time, as implied by FIGURE 14.

### 7.3.1 General Model of a Change-controlled Repository

FIGURE 15 shows an abstract model of a change-controlled repository, which consists of:

- version-controlled configuration items - instances of `VERSIONED_OBJECT<T>`;
- `CONTRIBUTION`s;
- an optional directory system of folders. If folders are used, the folder structure must also be versioned as a unit.

The actual type of links between the controlled repository and the other entities might vary - in some cases it might be association, in others aggregation; cardinalities might also vary. FIGURE 15 therefore provides a guide to the definition of actual controlled repositories, such as an EHR, rather than a formal specification for them.
8 Identification

8.1 General Scheme

The identification scheme described here requires two kinds of “identifier”: identifiers proper and references, or locators. An identifier is a unique (within some context) symbol or number given to an object, and usually written into the object, whereas a reference is the use of an identifier by an exterior object, to refer to the object containing the identifier in question. This distinction is the same as that between primary and foreign keys in a relational database system.

In the openEHR RM, identifiers and references are implemented with two groups of classes defined in the support.identification package. Identifiers of various kinds are defined by descendant classes of OBJECT_ID, while references are defined by the classes inheriting from OBJECT_REF. The distinction is illustrated in FIGURE 16. Here we see two container objects with OBJECT_IDs (since OBJECT_ID is an abstract type, the actual type will be another XXX_ID class), and various OBJECT_REFs as references.

8.2 Levels of Identification

In order to make data items locatable from the outside, identification is supported at 3 levels in openEHR, as follows:

- repository objects: entities such as the EHR (EHR IM) and VERSIONED_OBJECTs (Common IM) are identified uniquely;
- top-level content structures: versioned content structures such as COMPOSITION, EHR_STATUS, PARTY etc are uniquely identified by the association of the identifier of their containing VERSIONED_OBJECT and the identifier of their containing VERSION within the container;
- internal nodes: nodes within top-level structures are identified using paths.

Three kinds of identification are used respectively. For repository structures, meaningless unique identifiers (“uids”) are used. In most cases, the type HIER_OBJECT_ID will be used, which contains an instance of a subtype of the UID class, i.e. either an ISO OID or a IETF UUID (see http://www.ietf.org/rfc/rfc4122.txt; also known as a GUID). In general UUIDs are favoured since they require no central assignment and can be generated on the spot. A repository object can be referenced with an OBJECT_REF containing its identifier.

Versions of top-level structures are identified in a way that is guaranteed to work even in distributed environments where copying, merging and subsequent modification occur. The full identification of a version of a top-level structure is the globally unique tuple consisting of the uid of the owning
VERSIONED_OBJECT, and the two VERSION attributes version_tree_id and creating_system_id. The version_tree_id is a 1 or 3-part number string, such as “1” or for a branch, “1.2.1”. The creating_system_id attribute carries a unique identifier for the system where the content was first created; this may be a GUID, Oid or reverse internet identifier. A typical version identification tuple is as follows:

```
F7C5C7B7-75DB-4B39-9A1E-C0BA9BFDBDEC     -- id of VERSIONED_COMPOSITION
au.gov.health.rdh.ehr1               -- id of creating system
2                           -- current version
```

This 3-part tuple is known as a “Version locator” and is defined by the class OBJECT_VERSION_ID in the support.identification package. A VERSION can be referred to using a normal OBJECT_REF that contains a copy of the version’s OBJECT_VERSION_ID. The openEHR version identification scheme is described in detail in the change_control package section of the Common IM.

The last component of identification is the path, used to refer to an interior node of a top-level structure identified by its Version locator. Paths in openEHR follow an Xpath style syntax, with slight abbreviations to shorten paths in the most common cases. Paths are described in detail below.

To refer to an interior data node from outside a top-level structure, a combination of a Version locator and a path is required. This is formalised in the LOCATABLE_REF class in the change_control package section of the Common IM. A Universal Resource Identifier (URI) form can also be used, defined by the data type DV_EHR_URI (Data types IM). This type provides a single string expression in the scheme-space “ehr://” which can be used to refer to an interior data node from anywhere (it can also be used to represent queries; see below). Any LOCATABLE_REF can be converted to a DV_EHR_URI, although not all DV_EHR_URIs are LOCATABLE_REFS.

FIGURE 17 summarises how various types of OBJECT_ID and OBJECT_REF are used to identify objects, and to reference them from the outside, respectively.

![Diagram](image-url)
9 Archetyping

9.1 Overview

Under the two-level modelling approach, the formal definition of information structuring occurs at two levels. The lower level is that of the reference model, a stable object model from which software and data can be built. Concepts in the openEHR reference model are invariant, and include things like Composition, Section, Observation, and various data types such as Quantity and Coded text. The upper level consists of domain-level definitions in the form of archetypes and templates. Concepts defined at this level include things such as “blood pressure measurement”, “SOAP headings”, and “HbA1c Result”.

Archetypes are themselves instances of an archetype model, which defines a language in which to write archetypes. Archetypes are general-purpose, re-usable, and composable. They are used at runtime by building templates from them. A template is a tree of archetypes each of which constrains instances of various types in the reference model, i.e. Compositions, Section hierarchies, Entries and so on. Thus, while there are likely to be archetypes for such things as “biochemistry results” (an Entry archetype) and “SOAP headings” (a Section archetype), templates are used to put archetypes together to form whole Compositions in the EHR, e.g. for “discharge summary”, “antenatal exam” and so on. Templates correspond closely to screen forms and printed reports.

A template is used at runtime to create default data structures and to validate data input, ensuring that all data in the EHR conform to the constraints defined in the archetypes comprising the template. In particular, it conforms to the path structure of the archetypes, as well as their terminological constraints. Which archetypes were used at data creation time is written into the data, in the form of both archetype identifiers at the relevant root nodes, and archetype node identifiers - normative node names, which are the basis for paths. When it comes time to modify or query data, these archetype data enable applications to retrieve and use the original archetypes, ensuring modifications respect the original constraints, and allowing queries to be intelligently constructed.

All information conforming to the openEHR Reference Model (RM) - i.e. the collection of Information Models (IMs) - is “archetypable”, meaning that the creation and modification of the content, and subsequent querying of data is controllable by archetypes. Archetypes are themselves separate from the data, and are stored in their own repository. The archetype repository at any particular location will usually include archetypes from well-known online archetype libraries. Archetypes are deployed at runtime via templates that specify particular groups of archetypes to use for a particular purpose, often corresponding to a screen form.

9.2 Scope of Archetypes and Templates

All nodes within the top-level information structures in the openEHR RM are “archetypable”, with certain nodes within those structures being archetype “root points”. Each top-level type is always guaranteed to be an archetype root point. Although it is theoretically possible to use a single archetype for an entire top-level structure, in most cases, particularly for COMPOSITION and PARTY, a hierarchical structure of multiple archetypes will be used. This allows for componentisation and reusability of archetypes. When hierarchies of archetypes are used for a top-level structure, there will also be archetype root points in the interior of the structure. For example, within a COMPOSITION, ENTRY instances (i.e. OBSERVATIONS, EVALUATIONS etc) are almost always root points. SECTION instances are root points if they are the top instance in a Section structure; similarly for FOLDER instances within a directory structure. Other nodes (e.g. interior SECTIONs, ITEM_STRUCTURE
instances) might also be archetype root points, depending on how archetypes are applied at runtime to data. FIGURE 18 illustrates the application of archetypes and templates to data.

![FIGURE 18 How Archetypes apply to Data](image)

### 9.3 Archetype-enabling of Data

Archetype-enabling is achieved via inheritance into all concrete types in the RM of the class LOCATABLE from the package common.archetyped (see Common IM). The LOCATABLE class includes the attributes archetype_node_id and archetype_details. In the data, the former carries an identifier from the archetype. If the node in the data is a root point, it carries the multipart identifier of the generating archetype, and archetype_details carries an ARCHETYPED object, containing information pertinent to archetype root points. If it is a non-root node, the archetype_node_id attribute carries the identifier (known as an “at”, or “archetype term” code) of the archetype interior node that generated the data node, and the archetype_details attribute is void.

Sibling nodes in data can carry the same archetype_node_id in some cases, since archetypes provide a pattern for data, rather than an exact template. In other words, depending on the archetype design, a single archetype node may be replicated in the data.

In this way, each archetyped data composition in openEHR data has a generating archetype which defines the particular configuration of instances to create the desired composition. An archetype for “biochemistry results” is an OBSERVATION archetype, and constrains the particular arrangement of instances beneath an OBSERVATION object; a “problem/SOAP headings” archetype constrains SECTION objects forming a SOAP headings structure. In general, an archetyped data composition is any composition of data starting at a root node and continuing to its leaf nodes, at which point lower-level compositions, if they exist, begin. Each of the archetyped areas and its subordinate archetyped areas in FIGURE 18 is an archetyped data composition.

---

1. Note: care must be taken not to confuse the general term “composition” with the specific use of this word in openEHR and CEN EN 13606, defined by the COMPOSITION class; the specific use is always indicated by using the term “Composition”.
The result of the use of archetypes to create data in the EHR (and other systems) is that the structure of data in any particular top-level object conforms to the constraints defined in a particular composition of archetypes chosen by a template, including all optionality, value, and terminology constraints.

9.4 Archetypes, Templates and Paths

The use of archetypes and templates enables paths to be used ubiquitously in the openEHR architecture. Archetypes and templates have their own paths, constructed from attribute names and archetype node identifiers, in an Xpath-compatible syntax. These paths serve to identify any node in a template or archetype, such as the “diastolic blood pressure” ELEMENT node, deep within a “blood pressure measurement” archetype. Since archetype node identifiers are embedded into data at runtime, archetype paths can be used to extract data nodes conforming to particular parts of archetypes, providing a very powerful basis for querying. “Runtime” paths can also be constructed in data, consisting of more complex predicates (still in the Xpath style). Paths in openEHR are explained in details under Paths and Locators on page 34.
10 Paths and Locators

10.1 Overview

The openEHR architecture includes a path mechanism that enables any node within a top level structure to be specified from the top of the structure. The combination of a path and a Version identifier such as OBJECT_VERSION_ID forms a “globally qualified node reference” which can be expressed using LOCATABLE_REF. It can also be expressed in portable URI form as a DV_EHR_URI, known as a “globally qualified node locator”. Either representation enables any openEHR data node to be referred to from anywhere. This section describes the syntax and semantics of paths, and of the URI form of reference.

10.2 Paths

10.2.1 Basic Syntax

Paths in openEHR are defined in an Xpath1-compatible syntax which is a superset of the path syntax described in the Archetype Definition Language (ADL). The syntax is designed to be easily mappable to Xpath expressions, for use with openEHR-based XML.

The runtime path syntax used in locator expressions follows the general pattern of a path consisting of segments each consisting of an attribute name2, and separated by the slash (‘/’) character, i.e.:

\[
\text{attribute\_name} / \text{attribute\_name} / \ldots / \text{attribute\_name}
\]

Paths select the object which is the value of the final attribute name in the path, when going from some starting point in the tree and following attribute names given in the path. The starting point is indicated by the initial part of the path, and can be specified in three ways:

- **relative path**: path starts with an attribute name, and the starting point is the current point in the tree (given by some previous operation or knowledge);
- **absolute path**: path starts with a ‘/’; the starting point is the top of the structure;
- **movable path**: path starts with a movable path leader ‘//’ and is taken to be a pattern which can start anywhere in the data; the pattern is matched if an actual path can be found anywhere in the structure that matches the path given after the ‘//’ leader.

10.2.2 Predicate Expressions

Paths specified solely with attribute names are limited in two ways. Firstly, they can only locate objects in structures in where there are no containers, such as lists or sets. However, in any realistic data, including most openEHR data, list, set and hash structures are common. Additional syntax is needed to match a particular object from among the siblings referred to by a container attribute. This takes the form of a predicate expression enclosed in brackets (‘[]’) after the relevant attribute in a segment, i.e.:

\[
\text{attribute\_name} [\text{predicate expression}]
\]

The general form of a path then resembles the following:

---

1. See W3C Xpath 1.0 specification, 1999. Available at [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath).
2. In all openEHR documentation, the term “attribute” is used in the object-oriented sense of “property of an object”, not in the XML sense of named values appearing within a tag. The syntax described here should not be considered to necessarily have a literal mapping to XML instance, but rather to have a logical mapping to object-oriented data structures.
attribute_name / attribute_name [predicate expression] / ...
Here, predicate expressions are used optionally on those attributes defined in the reference model to be of a container type (i.e. having a cardinality of > 1). If a predicate expression is not used on a container attribute, the whole container is selected.

The second limitation of basic paths is that they cannot locate objects based on other conditions, such as the object having a child node with a particular value. To address this, predicate expressions can also be used to select an object on the basis of other conditions relative to the object, by including boolean expressions including paths, operators, values and parentheses.

The syntax of predicate expressions used in openEHR is a subset of the Xpath syntax for predicates with a small number of shortcuts. The general form of a predicate statement is a boolean-returning expression consisting of paths, values, operators and parentheses. In the current release of openEHR, it is expected that only very simple expressions will be used. The simplest such expression is to identify an object by its archetype_node_id value, which will be an ‘at’ code from an archetype; in other words, just to use the ADL archetype path against the runtime data. A typical ADL path is the following (applied to an Observation instance):

/data/events[at0003]/data/items[at0025]/value/magnitude
This path refers to the magnitude of a 1-minute Apgar total in an Observation containing a full Apgar result structure. In this path, the [atNNNN] predicates correspond to [@archetype_node_id = “atNNNN”] in standard Xpath, however, the shorthand form is used in openEHR as it is the only kind of predicate used in archetype paths. In openEHR runtime paths, archetype code predicates are also commonly used, and the same shortcut is allowed. However, runtime path predicates can also include other expressions (including the orthodox Xpath equivalent expression for the archetype node id shortcut), typically based on the value of some other attribute such as ELEMENT.name or EVENT.time. Combinations of the archetype_node_id and other such values are likely to be commonly used in querying, such as the following path fragment (applied to an OBSERVATION instance):

/data/events[at0007 AND time >= "24-06-2005 09:30:00"]
This path would choose Events in Observation.data whose archetype_node_id meaning is “summary event” (at0007 in some archetype) and which occurred at or after the given time. The following example would choose an Evaluation containing a diagnosis (at0002.1) of “other bacterial intestinal infections” (ICD10 code A04):

/data/items[at0002.1
    AND value/defining_code/terminology_id/value = “ICD10AM”
    AND value/defining_code/code_string = “A04”]

10.2.3 Paths within Top-level Structures
Paths within top-level structures strictly adhere to attribute and function names in the relevant parts of the reference model. Predicate expressions are needed to distinguish multiple siblings in various points in paths into these structures, but particularly at archetype “chaining” points. A chaining point is where one archetype takes over from another as illustrated in FIGURE 18. Chaining points in Compositions occur between the Composition and a Section structure, potentially between a Section structure and other sub-Section structures (constrained by a different Section archetype), and between either Compositions or Section structures, and Entries. Chaining might also occur inside an Entry, if archetyping is used on lower level structures such as Item_lists etc. Most chaining points correspond to container types such as List<T> etc, e.g. COMPOSITION.content is defined to be a List<CONTENT_ITEM>, meaning that in real data, the content of a Composition could be a List of Section structures. To distinguish between such sibling structures, predicate expressions are used, based on the archetype_id. At the root point of an archetype in data (e.g. top of a Section structure),
the archetype_id carries the identifier of the archetype used to create that structure, in the same manner as any interior point in an archetyped structure has an archetype_node_id attribute carrying archetype node_id values. The chaining point between Sections and Entries works in the same manner, and since multiple Entries can occur under a single Section, archetype_id predicates are also used to distinguish them. The same shorthand is used for archetype_id predicate expressions as for archetype_node_ids, i.e. instead of using @archetype_id = “xxxxx”, [xxxx] can be used instead.

The following paths are examples of referring to items within a Composition:

```plaintext
/content[@openEHR-EHR-SECTION.vital_signs.v1]/
  items[@openEHR-EHR-OBSERVATION.heart_rate-pulse.v1]/data/
  items[@at0003 AND time='2006-01-25T08:42:20']/data/items[@at0004]
/content[@openEHR-EHR-SECTION.vital_signs.v1]/
  items[@openEHR-EHR-OBSERVATION.blood_pressure.v1]/data/
  items[@at0006 AND time='2006-01-25T08:42:20']/data/items[@at0004]
/content[@openEHR-EHR-SECTION.vital_signs.v1]/
  items[@openEHR-EHR-OBSERVATION.blood_pressure.v1]/data/
  items[@at0006 AND time='2006-01-25T08:42:20']/data/items[@at0005]
```

Paths within the other top level types follow the same general approach, i.e. are created by following the required attributes down the hierarchy.

### 10.2.4 Runtime Paths and Uniqueness

Archetype paths are not guaranteed to be unique in data. However it will sometimes be necessary to be able to construct a unique path to any data item in real data. This can only be reliably done by using attributes other than archetype_node_id. Consider as an example the following OBSERVATION archetype:

```plaintext
OBSERVATION[@at0000] matches { -- blood pressure measurement
  data matches {
    HISTORY matches {
      events {1..*} matches {
        EVENT[@at0001] {0..1} matches { -- any event
          name matches {...}
          data matches {
            ITEM_LIST matches { -- systemic arterial BP
              count matches {2..*}
              items matches {
                ELEMENT[@at1100] matches { -- systolic BP
                  name matches {...}
                  value matches {magnitude matches {...}}
                }
                ELEMENT[@at1200] matches { -- diastolic BP
                  name matches {...}
                  value matches {magnitude matches {...}}
                }
              }
            }
          }
        }
      }
    }
  }
}
```

The following archetype path refers to the systolic blood pressure magnitude:

```plaintext
/data/events[@at0001]/data/items[@at1100]/value/magnitude
```
The codes \texttt{[atnnnn]} at each node of the archetype become the \texttt{archetype_node_ids} found in each node in the data.

Now consider an \texttt{OBSERVATION} instance (expressed here in dADL format), in which a history of two blood pressures has been recorded using this archetype:

\begin{verbatim}
<
  -- blood pressure measurement
  archetype_node_id = <[openEHR-EHR-OBSERVATION.bp_meas.v2]>
  name = <"xxx">
  data = <
    -- HISTORY
    events = <
      [1] = <
        archetype_node_id = <[at0001]>
        name = <"sitting">
        data = <
          -- ITEM_LIST
          items = <
            ["systolic"] = <
              archetype_node_id = <[at1100]>
              value = <magnitude = <120.0> ...>
            >
            ["diastolic"] = <....
              archetype_node_id = <[at1200]>
              value = <magnitude = <80.0> ...>
            >
          >
        >
      [2] = <
        archetype_node_id = <[at0001]>
        name = <"standing">
        data = <
          -- ITEM_LIST
          items = <
            ["systolic"] = <
              archetype_node_id = <[at1100]>
              value = <magnitude = <105.0> ...>
            >
            ["diastolic"] = <....
              archetype_node_id = <[at1200]>
              value = <magnitude = <70.0> ...>
            >
          >
        >
      >
    >
  >
>
\end{verbatim}

The archetype path mentioned above matches both systolic pressures in the recording. In many querying situations, this may be exactly what is desired. However, to uniquely match each of the systolic pressure nodes, paths need to be created that are based not only on the \texttt{archetype_node_id} but also on another attribute. In the case above, the \texttt{name} attribute provides uniqueness. Unique paths to the:

\begin{verbatim}
/data/events[at0001 AND name="sitting"]/data/items[at1100]/value/magnitude
/data/events[at0001 AND name="standing"]/data/items[at1100]/value/magnitude
\end{verbatim}

As a general rule, one or more other attribute values in the runtime data will uniquely identify any node in \texttt{openEHR} data. To make construction of unique paths easier, the value of the \texttt{name} attribute (inherited from the \texttt{LOCATABLE} class), is \texttt{required} to be unique with respect to the name values of sibling nodes. This has two consequences as follows:
• a guaranteed unique path can always be constructed to any data item in openEHR data using a combination of archetype_node_id and name values (as shown in the example paths above);
• the name value may be systematically defined to be a copy of one or more other attribute values. For example, in an EVENT object, name could clearly be a string copy of the time attribute.

10.3 EHR URIs

To create a reference to a node in an EHR in the form of a URI (uniform resource identifier), two elements are needed: the path within a top-level structure, and a reference to a top-level structure. These are combined to form a URI in an “ehr” scheme-space, obeying the following syntax:

\[
\text{ehr://top_level_structure_locator/path_inside_top_level_structure}
\]

Under this scheme, any object in any openEHR EHR is addressable via a URI. The openEHR data type DV_EHR_URI is designed to carry URIs of this form, enabling URIs to be constructed for use within LINKs and elsewhere in the openEHR EHR. (URIs of course are only one method of addressing or querying data in the EHR. Other querying syntaxes and functional interfaces will be developed and used over time.)

10.3.1 Locating Top-Level Structures

The first part of an EHR URI needs to identify a top-level structure. As described above, a Version locator can be used to do this. However, this is not the only way: various logical queries can also be used, e.g. “get the latest version from a given Versioned object matching...”’. For reasons of efficiency, the top-level structure locator part of the URI is also likely to include the EHR id, and possibly the EHR system id, even though neither of these are strictly needed for identification. Thus, the first part of an EHR URI might include the following:

• EHR id;
• EHR system id, depending on whether the EHR id is globally unique or not;
• Either:
  - version time, i.e. time baseline for retrieving versions, defaults to “now”;
  - identifier of particular Version container object, typically its uid;
• Or:
  - a Version identifier, i.e. {Version container object Uid; version_tree_id; creating_system_id}

For a number of reasons there is currently no standard syntax for encapsulating these parameters. Firstly, there is an issue to do with how EHRs will be identified in openEHR systems. Identifiers may be required to conform to local health jurisdiction requirements, or may not. If EHR identifiers are globally unique, or even nationally unique, then in theory the EHR system identifier can be dispensed with. However in a practical sense the identifier of the EHR system can only be dispensed with if there is a health information location service operating in the environment that can perform EHR id -> EHR system id mappings, in much the same way as the internet DNS converts logical domain names to IP addresses.

Another issue is whether it can be assumed that the version time baseline has already been established in some earlier call or service invocation, and that no version time information is needed.

Various syntax possibilities include:
Paths and Locators
Rev 1.0.2

- an Xpath-style syntax; this does not seem desirable as it implies hierarchical data containment structures that don’t exist and is likely to be confused with the path part of the URI;
- a web-services inspired functional syntax;
- a database-inspired query syntax.

Currently, a fairly typical URI query style of syntax is used, as shown in the following examples.

- **This path matches a VERSIONED_COMPOSITION:**
  ehr://rdh.health.gov.au?ehr=1234567&versioned_composition=87284370-2D4B-4e3d-A3F3-F303D2F4F34B

- **The following path matches the most recent VERSION<COMPOSITION> from a specified VERSIONED_COMPOSITION:**

- **The following path matches a COMPOSITION within VERSION 2 of a VERSIONED_COMPOSITION:**
  ehr://rdh.health.gov.au?ehr=1234567&version={F7C5C7B7-75DB-4b39-9A1E-C0BA9BFDBDEC::rdh.health.gov.au::2}&data

In these paths, the following pseudo-identifiers are used:

- **versioned_composition:** to indicate an instance of VERSIONED_COMPOSITION;
- **version_tree_id:** the version identifier of the VERSION within the tree structure of the owning VERSIONED_OBJECT;
- **latest_version:** a pseudo-identifier used to indicate a VERSION instance being the result of the call VERSIONED_COMPOSITION.latest_version;

Implementors and users of the current release of openEHR are encouraged to experiment and/or propose improved solutions to the locator requirement described in this section.
11 Relationship to Standards

The openEHR specifications make use of available standards where relevant, and as far as possible in a compatible way. However, for the many standards have never been validated in their published form (i.e. the form published is not tested in implementations, and may contain errors), openEHR makes adjustments so as to ensure quality and coherence of the openEHR models. In general, “using” a standard in openEHR may mean defining a set of classes which map it into the openEHR type system, or wrap it or express it in some other compatible way, allowing developers to build completely coherent openEHR systems, while retaining compliance or compatibility with standards. The standards relevant to openEHR fall into a number of categories as follows.

Standards by which openEHR can be evaluated
These standards define high-level requirements or compliance criteria which can be used to provide a means of normative comparison of openEHR with other related specifications or systems. The following ones have been used for this purpose so far:


Standards which have influenced the design of openEHR specifications
The following standards have influenced the design of the openEHR specifications:

- OMG HDTF Standards - general design
- CEN EN 13606:2006: Electronic Health Record Communication
- CEN HISA 12967-3: Health Informatics Service Architecture - Computational viewpoint

Standards which have influenced the design of openEHR archetypes
The following standards are mainly domain-level models of clinical practice or concepts, and are being used to design openEHR archetypes and templates.

- CEN HISA 12967-2: Health Informatics Service Architecture - Information viewpoint
- CEN ENV 13940: Continuity of Care.

Standards which are used “inside” openEHR
The following standards are used or referenced at a fine-grained level in openEHR:

- ISO 8601: Syntax for expressing dates and times (used in openEHR Quantity package)
- ISO 11404: General Purpose Datatypes (mapped to in openEHR assumed_types package in Support Information Model)
- HL7 UCUM: Unified Coding for Units of Measure (used by openEHR Data types)
- HL7v3 GTS: General Timing Specification syntax (used by openEHR Data types).
- some HL7v3 domain vocabularies are mapped to from the openEHR terminology.

Standards which require a conversion gateway
The following standards are in use and require data conversion for use with openEHR:

- CEN EN 13606:2005: Electronic Health Record Communication - near-direct conversion possible, as openEHR and CEN EN 13606 are actively maintained to be compatible.
- HL7v3 CDA: Clinical Document Architecture (CDA) release 2.0 - fairly close conversion may be possible.
- HL7v3 messages. Quality of conversion currently unknown due to flux in HL7v3 messaging specifications.
• **HL7v2 messages.** Experience in Australia indicates that importing of HL7v2 message information is relatively easy. Export from openEHR may also be possible.

**Generic Technology Standards**
The following standards are used or referenced in openEHR:

- ISO RM/ODP
- OMG UML 2.0
- W3C XML schema 1.0
- W3C Xpath 1.0
12 Implementation Technology Specifications

12.1 Overview

ITSs are created by the application of transformation rules from the “full-strength” semantics of the abstract models to equivalents in a particular technology. Transformation rules usually include mappings of:

- names of classes and attributes;
- property and function signature mapping;
- mapping of basic types e.g. strings, numerics;
- how to handle multiple inheritance;
- how to handle generic (template) types;
- how to handle covariant and contravariant redefinition semantics;
- the choice of mapping properties with signature $xxx:T$ (i.e. properties with no arguments) to stored attributes ($xxx:T$) or functions ($xxx():T$);
- how to express preconditions, postconditions and class invariants;
- mappings between assumed types such as `List<>`, `Set<>` and inbuilt types.

ITSs are being developed for a number of major implementation technologies, as summarised below. Implementors should always look for an ITS for the technology in question before proceeding. If none exists, it will need to be defined. A methodology to do this is being developed.

FIGURE 19 illustrates the implementation technology specification space. Each specification documents the mapping from the standard object-oriented semantics used in the openEHR abstract models, and also provides an expression of each of the abstract models in the ITS formalism.
# References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s) and Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>CORBAmed document: Person Identification Service. (March 1999). (Authors?)</td>
</tr>
<tr>
<td>9</td>
<td>CORBAmed document: Lexicon Query Service. (March 1999). (Authors?)</td>
</tr>
</tbody>
</table>
END OF DOCUMENT
Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Added class colour scheme.</td>
<td>T Beale</td>
<td>24 Dec 2005</td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2.1</td>
<td>Added section on functions and anchored types.</td>
<td>T Beale</td>
<td>24 Feb 2005</td>
</tr>
<tr>
<td>1.2</td>
<td>Added section on tooling.</td>
<td>T Beale</td>
<td>14 Feb 2005</td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.9</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 1.1.1 | CR-000041. Visually differentiate primitive types in openEHR documents.  
        | CR-000013. Rename key classes, according to CEN ENV 13606.  
        | Add explanation of qualified associations, existence, cardinality. | D Lloyd,  
        | D Kalra,  
        | T Beale | 04 Oct 2003 |
| 1.1   | Kestral Australia review. | G Grieve | 08 Mar 2003 |
| 1.0   | Adapted from openEHR EHR Reference Model document | T Beale | 10 May 2002 |

Acknowledgements
The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia, by Ocean Informatics, Australia, and by the CHIME department, University College London.
1 Introduction

1.1 Purpose
This document describes modelling method and tools of openEHR. It explains the usage of UML, how to read the openEHR specifications. The intended audience includes:

- Software development organisations using openEHR.

1.2 Overview
The openEHR Foundation provides specifications of health information systems and interoperability mechanisms in the form of formal, object-oriented models. These models are expressed in the OMG Unified Modelling Language (UML), along with detailed tabular descriptions. A formal textual expression is used to verify all models, ensuring that openEHR specifications are more than just paper. At the coarsest scale, the models are designed according to the ISO reference model for open distributed processing (RM/ODP). The openEHR models are divided into the Reference Model (RM), containing the information viewpoint, the Service Model (SM), containing the computational viewpoint, and the Archetype Model (AM), containing the formalisms for domain models, known as archetypes.

The models are a suitable starting point for system and interoperability software. Expressions in various implementation technologies are supplied by openEHR, known as Implementation Technology Specifications (ITSs). These are generated from the formal textual primary expression of the models.
2 The openEHR Modelling Environment

Since the primary users of the formal specifications in health information standards are software developers and information systems builders, it is crucial that the models presented are comprehensible and implementable by technical people. To ensure comprehensibility, the OMG standard UML 2.0 diagramming language has been used for graphical models. Detailed formal specifications of all classes are given, including class invariants and function pre- and post-conditions.

In order to ensure implementability, a tool-based environment is used for representation and manipulation of the models. The general approach is to have a single, authoritative “source” for any given artifact, and to use purpose-built converters to generate usable “views” of the source. The main way this is applied is that the core information and service model specifications (IMs and SMs) of the Reference Model are fully defined in object-oriented semantics, with implementation technology specifications (ITSs) being generated as views. For example, the XML-schema, Java interface classes, C# interface classes, and a relational schema view for the EHR are generated from the primary EHR model, which is expressed in formal textual UML 2.0-compliant semantics.

The primary expression of all object-oriented openEHR models is currently the ECMA-standardised Eiffel language (ECMA Eiffel page), as this the only textual formalism that closely approximates UML 2.0, and has tools available for it (including a free gnu implementation). The use of the Eiffel tools permit the core models to include all possible object-oriented semantics, including classes, attributes, functions, procedures, pre- and post-conditions, class invariants, multiple inheritance, genericity (“template” classes), agents (“delegates”), within a fully object-oriented type system (i.e. even basic types such as integer are instances of classes). Invariants are probably the single most important element of class specifications in any object model, since they indicate to the developer the valid instance structures in a system (for example, if a list attribute must be present, and if it is, whether the list can be empty and so on). All of these semantics can be validated, ensuring that the published specifications are much more than “just paper”, as is unfortunately common with many well-known published standards.

The primary models are used as the source for the published documentary form of the specifications, generally in Adobe PDF format. There is not considered to be any semantic difference between tool-based abstract model expressions and their documentary counterparts, i.e. there is no “mapping” or “conversion”.

The openEHR Modelling Environment

FIGURE 1 openEHR modelling environment

The primary expression of all object-oriented openEHR models is currently the ECMA-standardised Eiffel language (ECMA Eiffel page), as this the only textual formalism that closely approximates UML 2.0, and has tools available for it (including a free gnu implementation). The use of the Eiffel tools permit the core models to include all possible object-oriented semantics, including classes, attributes, functions, procedures, pre- and post-conditions, class invariants, multiple inheritance, genericity (“template” classes), agents (“delegates”), within a fully object-oriented type system (i.e. even basic types such as integer are instances of classes). Invariants are probably the single most important element of class specifications in any object model, since they indicate to the developer the valid instance structures in a system (for example, if a list attribute must be present, and if it is, whether the list can be empty and so on). All of these semantics can be validated, ensuring that the published specifications are much more than “just paper”, as is unfortunately common with many well-known published standards.

The primary models are used as the source for the published documentary form of the specifications, generally in Adobe PDF format. There is not considered to be any semantic difference between tool-based abstract model expressions and their documentary counterparts, i.e. there is no “mapping” or “conversion”.
The primary models are also losslessly translated to a UML-2.0 compliant XML instance form, from which all other views are generated. In theory, this intermediate form should be OMG XMI, but for various practical reasons it is not: not only are XMI documents massive and impossible for humans to read, but they do not correctly include pre- and post-conditions or invariants. However, the choice of the intermediate format may change in the future - the only requirements are that it be lossless with respect to the primary specifications, and that it be acceptable and processable by its users.
3  Formalisms

3.1  UML

The openEHR models are shown in UML (Unified Modelling Language) [5] and have been formally validated using the Eiffel language, which strangely, is still one of the only reliable tool for specifying and fully validating object-oriented models. UML is an industry-standard modelling language, which has been formally defined by the OMG. The openEHR models make heavy use of two powerful UML semantics, namely:

- Generic classes (“template classes” in C++)
- Contracts, i.e. pre-conditions, post-conditions, invariants (defined in the OMG Object Constraint Language, OCL)

The notation used in this document follows the UML version 1.3 (see [5]). The following sections describe the major semantic constructs in the class diagrams in this document. Refer to Meyer [8] for a definitive guide to object-oriented semantics.

3.1.1  Package

A collection of related classes, typically corresponding to one or more business objects, and grouped for convenient management of development. Packages may be nested hierarchically. Indicated graphically by a named blue rectangle containing classes.

3.1.2  Class

The primary construct in object-oriented modelling and software development. A class defines objects in terms of behaviour and state, or in more technical terms, routines and attributes. The class definition is the template for creating objects at runtime, which are instances of the class.

3.1.3  Inheritance

Inheritance is a relationship between classes in which the definition of the descendant (inheriting class) is based on the ancestor. The descendant may change the ancestor’s definition in certain ways, according to the rules of the formalism. Inheritance is not normally visible at runtime as a relationship between objects. A number of meanings can be assigned to inheritance relationships, including:

- Specialisation/generalisation
- Implementation re-use
- Facility inheritance (mixin classes)
- Taxonomic classification

3.1.4  Association

Association is a relationship between classes which describes a runtime relationship between objects. Its cardinality may be single (1:1) or multiple (1:N).

A particular kind of association between classes indicates the logical part-of relationship. There are two recognised variants of this, namely composition, or containment-by-value, and aggregation meaning a logical part-of relationship. The use of these different types of association in the model is detailed below.
3.2 Other Alternatives

Numerous alternatives were considered both in the original GEHR project, and for openEHR, including the following:

- **OMG IDL**: the OMG’s IDL language lacks assertions and generic types, and its type model is inconsistent (basic “types” are not the same as constructed types, due to the influence of C);

- **Rumbaugh/Booch/etc notations**: none of these notations are formal, and all lack assertions. In any case, they have been superseded by UML;

- **SGML/XML**: SGML is overly complex, and very document-oriented.

- **XML-schema**: not well adapted to information modelling (cf information representation) because it is purely data-oriented, and missing a number of important semantics, namely assertions, generic types and multiple inheritance.

- **Z, Object Z, B**: these are worthy of future consideration. Their use now is prevented mainly by a lack of industrial strength tools.
4 Modelling Guide

4.1 General Principles

One crucial point to understand about modelling is that the semantics of all definitions in a model constitute statements about the informational (or behavioural) entities defined by the relevant classes, and no more. Many modellers make the mistake of entering into torturous discussions about the semantics of real world objects based on the arrangement of classes or relationships in a model, when in fact the argument should be the other way around - any model is a formalisation and abstraction, potentially of real world entities, and its meaning does not extend beyond itself. Thus, any concept in a model, such as defined by the type QUANTITY should not be understood as being a description of quantities in the real world, but a formal, abstract model of a concept called “quantity” as agreed by the modellers.

4.2 Naming

Class names are in upper case, with underscore separators, enabling them to be easily identified and read. In almost all cases, the full english word has been used. Class feature names (i.e. attribute and method names) are in lower case, underscore separated.

This style choice may surprise some developers, and has been done in purpose for a number of reasons:

- for better readability;
- to make it clear that the specification is an abstract one, and to prevent confusion with programming languages ITSSs;
- to allow the use of mixed case class names for the assumed types of UML, such as String, Integer, List<T> and so on; these contrast with classes defined by openEHR, such as SECTION and ENTRY, making it very clear what openEHR has defined versus what it has assumed.

The names used in the abstract specifications are transformed by tools into the preferred idiom of each target formalism; the rules for doing so are described in each ITS.

All names have been chosen with implementors and other people in mind who will deal with technical modelling, rather than users. In almost all cases, users will never see the names used in the reference or archetype models. The exception is archetype editor tools which would normally show the class names of instances of the archetype model which are being created; it is assumed that users of this tool will have a basic technical understanding of the reference and archetype models.

4.3 Operators

Three classes of operator are used:

- infix operators, i.e. any binary operator which appears between the operands, e.g. “+” in the expression “X + Y”
- prefix operators, i.e. any unary operator appearing before the operand, e.g. “-” in “-5”
- postfix operators, i.e. any unary operatory appearing after the operand, e.g. “!” in “x!” (factorial).
4.4 Types

The reference model can be thought of as consisting of a number of classes which fulfill one of two purposes. The first category includes those which represent concretely-modelled concepts like “revision history entry” or “transaction”, while the second includes those whose job it is to represent generic data structures, used to express clinical data whose specific form is defined by archetypes, rather than by the reference model. The general form of the latter can best be understood as structures of name/value pairs, where all nodes in the structure have names, and leaf nodes have values as well. There are accordingly two kinds of “datatypes” used in the model: one for the attributes of all classes, and the other for the values in the clinical name/value structures. These latter are known as “data value types”, whereas the former are known as “attribute types”. Instances of data value types are the only allowable values in the generic information structures.

In addition to types defined in the model, a number of basic types are assumed in the modelling formalism, which are globally understood in the same (or compatible) ways in all implementation formalisms. These are:

- Character (members of a character set)
- String (strings of printable characters)
- Integer (integer numbers)
- Real (real numbers)
- Double (double precision floating point real numbers)
- Boolean (two-valued entities)
- Array<T> (physical container of items indexed by number)
- List<T> (implied order, non-unique membership)
- Set<T> (no order, unique membership)

4.4.1 Data Value Types

Data value types are characterised by being explicitly modelled and inheriting from the abstract class DATA_VALUE. The names of all of these types are prefixed with “DV_” to differentiate them from types of the same names which may occur in particular implementation technologies, thus DV_DATE rather than DATE and so on. Types which are notionally one of the standard basic types have a specific model. For example, the notional “string” type is modelled as the data value type DV_PLAIN_TEXT.

Data value types are the only types which can be used as data values, e.g. as the type of the ELEMENT.value attribute in the openEHR EHR reference model, or other similar places where the type DATA_VALUE is specified.

![FIGURE 2 rm.data_structures.representation Package](image-url)
4.4.2 Attribute Types

Types which can be used for other attributes in model classes include any standard basic type, or any of the data value types. For example, if a string is needed, the class STRING may be used, unless special features of DVPLAINTEXT are required. If a date/time is needed however, since there is no guaranteed standard type for this, the data value type DV_DATE_TIME must be used.

4.4.3 Existence and Cardinality

Existence of attributes is indicated by brackets after the attribute name inside a class box. Possible values are: [0..1], [1], meaning optional and mandatory, respectively. For attributes of container types such as List<T>, existence of the whole container is shown the same way. Cardinality of the container is shown by including the container type explicitly.

<table>
<thead>
<tr>
<th>PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>name [1]: String</td>
</tr>
<tr>
<td>address [0..1]: List&lt;String&gt;</td>
</tr>
</tbody>
</table>

FIGURE 3 Attribute Existence and Cardinality

4.5 Relationships

Relationships between classes in the reference model are of three logical types, described below.

4.5.1 Composition

Composition indicates the part/sub-part relationship where the sub-part can have no meaningful existence outside of the whole, or, put another way, the lifetime of the part is controlled by the whole. For example, in the openEHR EHR RM, the class COMPOSITION has as a subpart ACCESS_CONTROL, illustrated in FIGURE 4. All objects contained within a single “business object”, i.e. Strings, Integers and other leaf types are always related to the containing object by composition.

In UML, composition is indicated by a black diamond on the class representing the whole. A “part” object can only be in a composition relationship with one “whole” object, i.e. a given instance cannot be part-of multiple wholes.

<table>
<thead>
<tr>
<th>COMPOSITION</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ACCESS_CONTROL</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CODED TERM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rubric: String</td>
</tr>
<tr>
<td>concept_code: String</td>
</tr>
</tbody>
</table>

FIGURE 4 Examples of Composition

Semantically, composition corresponds to physical containment by value. Movement or deletion of the whole causes movement or deletion respectively of the part.

4.5.2 Aggregation

Aggregation indicates a logical part/sub-part relationship, where the sub-part can meaningfully exist on its own, i.e. does not need to be deleted if the parent whole is deleted. Consider by way of example
the relationship between \textsc{health\_care\_facility} and \textsc{health\_care\_professional} illustrated in FIGURE 5. The difference in semantics with respect to composition is that aggregation parts and wholes represent business objects (e.g. \textsc{hospital} and \textsc{person}), whereas the part objects of composite relationships represent fine-grained constituents inside a business object (e.g. \textsc{person} and \textsc{person\_name}).

![Example of Aggregation](image)

**FIGURE 5** Example of Aggregation

In UML, aggregation is indicated by a white diamond on the class representing the whole and a key shown on the part, meaning that the whole class contains a key referring to the part class. Movement or deletion of the whole may occur without movement or deletion of the part.

A sensible definition of the semantics of aggregation has historically been, and remains, problematic for many modellers. Various books on UML including “UML Distilled” [5], and indeed the authors of UML themselves have noted the confusion\(^1\), and done little to clear it up. Consequently, in some publications, the aggregation relationship has the semantics of allowing a “part” to be a part-of more than one whole. We see this as an error for a number of reasons.

- Firstly, there is no sensible understanding in natural language for the concept of something that is part of more than one whole.
- When the semantics of changing the part are investigated, it is normally found that a change to the part, seen as part-of one whole is not expected to cause a change in the same part seen as part-of another whole. If the change should indeed occur in all wholes, then the whole/part relationships are associations, and do not represent the part-of relationship at all. If changes are not meant to be global to all wholes, then distinct (possibly initially identical) instances of the part must be part-of each whole.
- In some models, aggregation is used in an attempt to represent “re-use”. However, re-use is not a meaningful modelling concept, although it is a meaningful implementation concept\(^2\). The only reasonable modelling interpretation of “re-use” would be that a part is part-of one whole, and there are other similar wholes that have (or will have at a later point in time) a part which is identical to the existing part. In this case, the proper interpretation of aggregation is that each whole has a part, and that there are also constraints or operations (such as copy) which guarantee that the parts of certain wholes are all identical in value to each other.

---

1. Jim Rumbaugh says of aggregation “think of it as a modelling placebo” (Rumbaugh, Jacobsen and Booch 1999) [3]. Martin Fowler calls it “one of my biggest betes noires”. Clearly, aggregation is not well understood by the “experts” [5].
2. This is well known as the “flyweight” pattern described in [6]. In the UML diagram for this pattern, an aggregation relationship appears between a flyweight-factory and the flyweight (shared) object; associations appear between the logical “owners” and the factory-generated flyweights.
Consequently, in this document aggregation semantics are defined such that a “part” object can only be in an aggregation relationship with one “whole” object, i.e. a given instance cannot be part-of multiple wholes.

### 4.5.3 Association

Association indicates any other kind of relationship in which instances of both classes are completely meaningful in themselves. Indicated by no diamond in UML.

### 4.5.4 Qualified Association

One kind of association which occurs quite commonly is the “qualified association”. In contrast with normal associations which are “direct” (i.e. object to object), qualified associations are by symbolic reference, where the reference is in the form of an attribute value from the target class. FIGURE 6 illustrates the qualified association, and shows an equivalent single class below it. The qualified association is most commonly used when objects of the target class will each have a unique id which can be referenced from elsewhere, in the manner of a primary/foreign key in relational systems (here the foreign key is the attribute `bar_id: String` in the class Foo.

![Qualified Association Diagram](image)

**FIGURE 6** Qualified Association

### 4.6 Functions

Functions are understood in openEHR in the object-oriented sense as “computed features having a return type, and not causing side-effects in the object on which they are called”. Functions are used in various places to define relevant interface, to support the expression of invariants, or to express computation of derived properties, such as extracting the logical pieces from a URI string.

### 4.7 Anchored Types

An object-oriented feature used a couple of times in the openEHR specifications is that of “anchored types”. An example of such a type is shown in the following UML.

![Anchored Types Diagram](image)

**FIGURE 7** pseudo-UML for Anchored Types
In this figure, an infix function ‘<’ is defined on the abstract type ORDERED. The signature has the parameter other, of type “like Current”. This syntax has been adopted from the Eiffel language because it is so useful in specification. Its effect is to say that in every subtype of ORDERED there is a function ‘<’ whose other argument is of the same type as the subtype - with no actual repeated definitions required to do this. This clearly saves on code, reduces errors of repetition. Although it is not available in UML, or in many languages, it is used in the specifications to reduce the repetition, and improve clarity. Mapping to implementation formalisms is easy: simply define the initial signature as having the same type as the type on which it is defined (ORDERED in this above example), and redefine the signature appropriately down the inheritance tree.

4.8 Constraints and Contracts

Constraints are written in a order predicate logic based on the OMG’s Object Constraint Language (OCL), with some differences due to problems in the current definition of OCL.

The keywords used are:

- **require**: routine precondition
- **ensure**: routine postcondition
- **invariant**: class invariant
- **and**, **or**, **and then**, **or else**, **implies**, **xor**: Boolean operators

The various kinds of constraints together form the “contract” of a class, that is the conditions under which its instances interact with instances of other classes (including itself). The following sections describe the three constraint types. See Meyer [8] and Kilov [7] for an explanation of contracts.

4.8.1 Pre-conditions

Pre-conditions are introduced with the keyword **require**, and consist of a first-order predicate logic expression evaluating to True or False. A pre-condition represents the truth condition which must be upheld by the caller of a routine to ensure the correct functioning of the routine, i.e. it is a condition assumed to be true by the routine. If a pre-condition is violated, the caller is in the wrong.

4.8.2 Post-conditions

Post-conditions are introduced with the keyword **ensure**, and consist of a first-order predicate logic expression evaluating to True or False. A post-condition represents the truth condition which must be upheld by a routine, i.e. it is a condition guaranteed to be true by the routine to the caller. If a post-condition is violated, the called routine is in the wrong.

4.8.3 Invariants

Invariants consist of first order predicate logic statements which apply to the whole class. The meaning is that for every instance of the class, the condition is true at all times, apart from mid-execution of a routine. In other words, object invariants are always true at the points in time when they are accessible to other objects - including prior to calling a routine, and upon exit. If an invariant fails, there is an error in the design of the class. Invariants must be satisfied upon completion of any creation routine.

4.9 Special Types

The type Any is assumed as the parent type of all other types, and is the type on which basic operators of equality and assignment are defined. See the Support Reference Model for details.
4.10 Special Instances

The following special instances are indentified in constraints.

- **Result** - the result of any function. “Result” is treated like a normal variable whose type is the return type of the function;
- **Current** - the current object. Synonymous with “self” in some languages.
- **Void** - the empty pseudo-object; conforms to any type. Means the absence of an instance. Synonymous with “null” in many languages.
# 5 Class Descriptions

Classes in the openEHR models are formally described in tabular form. Three colour schemes are used, as follows:

- **Cyan**: reference model classes
- **Light green**: archetype model classes

Reference model classes are defined in the form shown below. The various meanings of each section are indicated in this example table.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CLASS_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Description of purpose of class in model and information based on it.</td>
</tr>
<tr>
<td>Use</td>
<td>Particular uses of class in the model, or instances of the class in data.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Potential expected misuses of the class, usually based on common misuses or misconceptions of the name of the class.</td>
</tr>
<tr>
<td>CEN</td>
<td>Correspondence to CEN ENV 13606 part 1 - part 4 concepts. These standards were published by CEN in 2000, and can be found on <a href="http://www.centc251.org">http://www.centc251.org</a>.</td>
</tr>
<tr>
<td>Synapses</td>
<td>Correspondence to concepts in the Synapses, SynEx and EHCR-support Action models, produced in various EC-funded (4th framework) post-original GEHR projects. Some of this work is available at <a href="http://www.chime.ucl.ac.uk">http://www.chime.ucl.ac.uk</a>.</td>
</tr>
<tr>
<td>GeHR</td>
<td>Correspondence to Australian GEHR models as originally published on <a href="http://www.gehr.org">http://www.gehr.org</a> (now defunct); see specifications at <a href="http://www.openehr.org/gehr_australia/t_aus_gehr.htm">http://www.openehr.org/gehr_australia/t_aus_gehr.htm</a>.</td>
</tr>
<tr>
<td>HL7v3</td>
<td>Correspondence to concepts in HL7 version 3 models, as published in various ballots at <a href="http://www.hl7.org">http://www.hl7.org</a>.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>attr_1: SOME_TYPE</td>
</tr>
<tr>
<td>1..1</td>
<td>attr_n: SOME_TYPE</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>func_1(some_args: SOME_TYPE): SOME_TYPE</td>
<td>Description of this function</td>
</tr>
<tr>
<td>require</td>
<td>precondition</td>
</tr>
<tr>
<td>ensure</td>
<td>postcondition</td>
</tr>
<tr>
<td>func_n(some_args: SOME_TYPE): SOME_TYPE</td>
<td>Description of this function</td>
</tr>
</tbody>
</table>
Invariants

Class invariants. Each mandatory attribute must have an invariant of the form:

\[ \text{Attr}_1 \text{ exists}: \text{attr} \neq \text{Void} \]

Other invariants may be stated. All invariants have to be true before and after calls to routines (procedures or functions) made from outside an object.

Archetype classes are defined using tables like the following.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>Archetype CLASS XXX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>xxxxx</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>1..1</td>
<td>aaa: T</td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>fff: T</td>
</tr>
<tr>
<td>Invariant</td>
<td>Aaa_valid: aaa /= Void</td>
</tr>
</tbody>
</table>
A References


4  Fowler M. Analysis Patterns: Reusable Object Models
Addison Wesley 1997

5  Fowler M, Scott K. UML Distilled (2nd Ed.)
Addison Wesley Longman 2000


END OF DOCUMENT
openEHR Architecture

Support Terminology

Editors: {T Beale, S Heard}¹, {D Kalra, D Lloyd}²

Revision: 0.9

Pages: 25

---

1. Ocean Informatics Australia
2. Centre for Health Informatics and Multi-professional Education, University College London

© 2005-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

**Founding Chairman**
- David Ingram, Professor of Health Informatics, CHIME, University College London

**Founding Members**
- Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

**email**: info@openEHR.org **web**: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.

2. You may read and print the document for private, non-commercial use.

3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.

4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).

5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"

6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.

7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
</table>

Acknowledgements
The work reported in this paper has been funded by The University College, London and Ocean Informatics, Australia.
Table of Contents
1 Introduction

1.1 Purpose
This document describes the openEHR Support Terminology, which defines the vocabularies needed for the openEHR Reference, Archetype and Service models. This terminology is not considered to be in the same space as externally defined terminologies such as Snomed-ct, ICDx etc, but rather a part of the infrastructure of the openEHR architecture. The audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations developing EHR systems;
- Academic groups studying the EHR;
- The open source healthcare community.

1.2 Related Documents
Prerequisite documents for reading this document include:

- The openEHR Reference Model documents.

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

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

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

Blue text indicates sections under active development.

1.4 Peer review
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.

1.5 Conformance
Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Terminology

2.1 Overview

This document provides a documentary expression of the openEHR terminology, which consists code sets and term lists, which provide values for the dozen or so structural attributes in the openEHR Reference Model. The computable form of this terminology is available in the computable part of the openEHR specification repository, and should always be considered the definitive expression, rather than this document.

There are two types of coded terms used. The first are ‘proper’ coded terms, where each code is a concept identifier, for which there can be a rubric and description in multiple languages. In other words, they way of ‘saying’ the concept is dependent on the language one is working in. Most clinical terminologies are in this category, e.g. ICD10, ICPC. Terminologies in this category are modelled in openEHR by the TERMINOLOGY class, and by terms expressed as instances the DV_CODED_TEXT class, each of which has as an attribute a defining CODE PHRASE - the actual code.

The second category is codes which are self-defining, and which do not have separate rubrics. The ISO country and language codes are examples of this, as are code groups for such concepts as ‘integrity check algorithm names’. This category is modelled in openEHR by the CODE_SET which is made up of CODE PHRASES. Value sets which cannot meaningfully be translated into other languages and which do not have definitions beyond their code value are usually candidates for being a code set rather than a terminology group.

Both code set definitions and terminology groups provide mappings to other recognised terminologies or vocabularies. Given that the attributes defined here are mostly structural attributes (i.e. predefined in the openEHR Reference Model), mappings tend to be to terms in vocabularies defined by standards organisations such as CEN and HL7, rather than large clinical vocabularies such as ICD10 (WHO). OpenEHR does not specify the use of these vocabularies.

2.2 Code Sets

Code sets are not shown in full here, since their codes are derived from resources published by external authorities; however, the openEHR code-set databases contain the full set of codes in each case.

2.2.1 Languages

This ISO code set defined by the ISO 639 standard consists of the “alpha-2” form of names of languages. This does not cover all languages, whereas ISO 639 “alpha-3” covers many more languages of cultural or indigenous interest, but which nevertheless are unlikely to be supported by current software or operating systems. See http://www.loc.gov/standards/iso639-2/langhome.html.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ab&quot;</td>
<td>Abkhazian</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;bg&quot;</td>
<td>Bulgarian</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;zh&quot;</td>
<td>Chinese</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;en&quot;</td>
<td>English</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;es&quot;</td>
<td>Spanish</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;fr&quot;</td>
<td>French</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;de&quot;</td>
<td>German</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;it&quot;</td>
<td>Italian</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;ja&quot;</td>
<td>Japanese</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>&quot;zh&quot;</td>
<td>Chinese</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
### 2.2.2 Countries
This ISO code set defined by the ISO 3166 standard consists of 2-character names of countries and country subdivisions. For a definitive online rendition see [http://www.unicode.org/unicode/online-dat/countries.html](http://www.unicode.org/unicode/online-dat/countries.html).

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;af&quot;</td>
<td>&quot;Afghanistan&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;al&quot;</td>
<td>&quot;Albania&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.3 Character Sets
This IANA (Internet Naming Authority) code set consists of the names of recognised character sets. See [http://www.iana.org/assignments/character-sets](http://www.iana.org/assignments/character-sets) for authoritative source.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO-10646-UTF-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISO_8859-3:1988</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 2.2.4 Media Types
This IANA (Internet Naming Authority) code set consists of the names of MIME media types. See [http://www.iana.org/assignments/media-types/text/](http://www.iana.org/assignments/media-types/text/) for authoritative source.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;text/plain&quot;</td>
<td>Plain text encoded according to RFC3676</td>
<td>HL7_MediaType::14826</td>
</tr>
<tr>
<td>&quot;text/html&quot;</td>
<td>HTML text encoded according to RFC2854</td>
<td>HL7_MediaType::14828</td>
</tr>
<tr>
<td>&quot;text/richtext&quot;</td>
<td>Rich text encoded according to RFC2046</td>
<td>HL7_MediaType::14831</td>
</tr>
<tr>
<td>&quot;text/sgml&quot;</td>
<td></td>
<td>HL7_MediaType::14829</td>
</tr>
<tr>
<td>&quot;text/rfc822-headers&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;text/xml&quot;</td>
<td></td>
<td>HL7_MediaType::14830</td>
</tr>
<tr>
<td>&quot;audio/basic&quot;</td>
<td></td>
<td>HL7_MediaType::14836</td>
</tr>
<tr>
<td>&quot;audio/mpeg&quot;</td>
<td></td>
<td>HL7_MediaType::14837</td>
</tr>
<tr>
<td>&quot;application/pdf&quot;</td>
<td></td>
<td>HL7_MediaType::14833</td>
</tr>
<tr>
<td>&quot;application/msword&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2.5 Compression algorithms
This code set consists of the names of algorithms used to compress data, and is drawn from HL7’s CompressionAlgorithms domain.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>“compress”</td>
<td>Original UNIX compress algorithm and file format using the LZC algorithm (a variant of LZW).</td>
<td>HL7_CompressionAlgorithm::10624</td>
</tr>
<tr>
<td>“other”</td>
<td>Some other type of compression; might be retrievable upon direct inspection of data.</td>
<td></td>
</tr>
</tbody>
</table>

2.2.6 Integrity check algorithms
This code set consists of the names of algorithms used to generate hashes for the purpose of integrity checks on data; its initial values are drawn from the HL7 IntegrityCheckAlgorithm domain.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>“SHA-1”</td>
<td>Secure hash algorithm - 1. Defined in FIPS PUB 180-1: Secure Hash Standard. As of April 17, 1995.</td>
<td>HL7_IntegrityCheckAlgorithm::17386</td>
</tr>
<tr>
<td>“SHA-256”</td>
<td>secure hash algorithm - 256. Defined in FIPS PUB 180-2: Secure Hash Standard</td>
<td>HL7_IntegrityCheckAlgorithm::17387</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

2.3 Vocabularies and Terminologies

2.3.1 Attestation Reason
This vocabulary codifies attestation statuses of Compositions or other elements of the health record, and is drawn from the HL7 ParticipationSignature domain, as used in CDA.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>“signed”</td>
<td>The attested information has been signed by its signatory.</td>
<td>HL7_ParticipationSignature::10284</td>
</tr>
<tr>
<td>???</td>
<td>“witnessed”</td>
<td>This attested information has been witnessed by the signatory.</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.3.2 Audit Change Type
This vocabulary codifies the kinds of changes to data which are recorded in audit trails.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>249</td>
<td>“creation”</td>
<td>Change type was creation.</td>
<td>HL7_CDA: CEN:</td>
</tr>
<tr>
<td>250</td>
<td>“amendment”</td>
<td>Change type was amendment, i.e. correction of the previous version.</td>
<td>HL7_CDA: CEN:</td>
</tr>
<tr>
<td>251</td>
<td>“modification”</td>
<td>Change type was update of the previous version.</td>
<td>HL7_CDA: CEN:</td>
</tr>
<tr>
<td>252</td>
<td>“synthesis”</td>
<td>Change type was creation synthesis of data due to conversion process, typically a data importer.</td>
<td>HL7_CDA: CEN:</td>
</tr>
<tr>
<td>523</td>
<td>“deleted”</td>
<td>Change type was logical deletion.</td>
<td>HL7_CDA: CEN:</td>
</tr>
<tr>
<td>253</td>
<td>“unknown”</td>
<td>Type of change unknown.</td>
<td>HL7_CDA: CEN:</td>
</tr>
</tbody>
</table>

2.3.3 Composition Category
This vocabulary codifies the values of the category attribute of the COMPOSITION class in the rm.composition package.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>431</td>
<td>“persistent”</td>
<td>This Composition contains information which remains valid for (more or less) the life of the EHR. Typical persistent Compositions include “family history”, “problem list”, “current medications”, and “vaccination history”. The usual change type when creating a new version of a persistent composition is “modification”.</td>
<td></td>
</tr>
<tr>
<td>433</td>
<td>“event”</td>
<td>This composition pertains to a point in time or brief episode. Change types may usually be “modification” or “</td>
<td></td>
</tr>
</tbody>
</table>

2.3.4 Event Math Function
This vocabulary codifies mathematical functions of non-instantaneous events.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>145</td>
<td>“minimum”</td>
<td>Value of the interval-event is the minimum value of the discrete events which the interval-event summarises.</td>
<td></td>
</tr>
</tbody>
</table>
2.3.5 Instruction State Machine (ISM) States

This vocabulary codifies the names of the states in the standard Instruction state machine, documented in the openEHR EHR Information model (Entry section).

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>524</td>
<td>“initial”</td>
<td>The instruction is recorded but no state is determined</td>
<td></td>
</tr>
<tr>
<td>526</td>
<td>“planned”</td>
<td>The instruction is planned</td>
<td></td>
</tr>
</tbody>
</table>
2.3.6 Instruction State Machine (ISM) Transitions

This vocabulary codifies the names of the transitions in the standard Instruction state machine, documented in the openEHR EHR Information model (Entry section).

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>527</td>
<td>“postponed”</td>
<td>The instruction has been postponed - it had not be commenced</td>
<td></td>
</tr>
<tr>
<td>528</td>
<td>“cancelled”</td>
<td>The instruction has been cancelled - it had not been commenced and will not commence in the future</td>
<td></td>
</tr>
<tr>
<td>529</td>
<td>“scheduled”</td>
<td>The instruction has been scheduled to be carried out at a particular time</td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>“active”</td>
<td>The instruction is currently being carried out</td>
<td></td>
</tr>
<tr>
<td>530</td>
<td>“suspended”</td>
<td>The instruction is suspended, it has been activated but is not active at present. It could be active again in the future.</td>
<td></td>
</tr>
<tr>
<td>531</td>
<td>“aborted”</td>
<td>The instruction is aborted, it has been activated but ceased before it has been completed and will not be restarted in the future.</td>
<td></td>
</tr>
<tr>
<td>532</td>
<td>“completed”</td>
<td>The instruction has been completed</td>
<td></td>
</tr>
<tr>
<td>533</td>
<td>“expired”</td>
<td>The instruction has expired, timed out - and assumed to have either been cancelled, aborted or completed</td>
<td></td>
</tr>
<tr>
<td>535</td>
<td>“initiate”</td>
<td>Initiate the planning of the Instruction.</td>
<td></td>
</tr>
<tr>
<td>536</td>
<td>“plan step”</td>
<td>Any step in the planned state of the Instruction, e.g. signing, approving.</td>
<td></td>
</tr>
<tr>
<td>537</td>
<td>“postpone”</td>
<td>Put a planned Instruction on hold, while still in the planning stage, i.e. before it has been booked or started.</td>
<td></td>
</tr>
<tr>
<td>538</td>
<td>“restore”</td>
<td>Restore a previously postponed Instruction back to the planned state.</td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>“cancel”</td>
<td>Cancel a planned Instruction, before it is booked or started.</td>
<td></td>
</tr>
<tr>
<td>542</td>
<td>“postponed step”</td>
<td>Any step in the postponed state of the Instruction.</td>
<td></td>
</tr>
<tr>
<td>539</td>
<td>“schedule”</td>
<td>Where booking is required, book the activities in the Instruction in a scheduling system.</td>
<td></td>
</tr>
<tr>
<td>540</td>
<td>“start”</td>
<td>Start executing the activities in the Instruction, e.g. commence drug administration course.</td>
<td></td>
</tr>
</tbody>
</table>
### Terminology: openehr Group_name("en"): “ISM transitions”

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>541</td>
<td>“do”</td>
<td>Do the activities in the Instruction in one go, taking the state machine directly from the planned to the completed state. Used for Instructions whose activities are instantaneous in the practical sense, e.g. a single vaccination, single tablet.</td>
<td></td>
</tr>
<tr>
<td>543</td>
<td>“active step”</td>
<td>Any step taken during the active phase of the Instruction, e.g. nurse’s observation, adjustment of dose.</td>
<td></td>
</tr>
<tr>
<td>544</td>
<td>“suspend”</td>
<td>Suspend the activities from the active phase, with the possibility of later resumption.</td>
<td></td>
</tr>
<tr>
<td>545</td>
<td>“suspended step”</td>
<td>Any step taken in the suspended state, e.g. nurse’s observation, pathology test to determine if the Instruction should be resumed, remain suspended or aborted.</td>
<td></td>
</tr>
<tr>
<td>546</td>
<td>“resume”</td>
<td>Resume the Instruction from the suspended state.</td>
<td></td>
</tr>
<tr>
<td>547</td>
<td>“abort”</td>
<td>Abort the Instruction, i.e. stop its execution permanently after it has started.</td>
<td></td>
</tr>
<tr>
<td>548</td>
<td>“finish”</td>
<td>Finish performing the Instruction, taking it to the completed state.</td>
<td></td>
</tr>
<tr>
<td>549</td>
<td>“time out”</td>
<td>Time out has occurred, taking the Instruction from some previous state into the expired state.</td>
<td></td>
</tr>
<tr>
<td>540</td>
<td>“notify aborted”</td>
<td>Occurs when notification of Instruction having been aborted is received after expiry.</td>
<td></td>
</tr>
<tr>
<td>551</td>
<td>“notify completed”</td>
<td>Occurs when notification of Instruction having been completed is received after expiry.</td>
<td></td>
</tr>
<tr>
<td>552</td>
<td>“notify cancelled”</td>
<td>Occurs when notification of Instruction having been cancelled is received after expiry.</td>
<td></td>
</tr>
</tbody>
</table>

### 2.3.7 Measurable Properties

This vocabulary codifies purposes for physical properties corresponding to formal unit specifications, and allows comparison of Quantities with different units but which measure the same property. The vocabulary values are taken from:

- CEN ENV 12435 - “Medical Informatics - Expression of results of measurements in health sciences”
- HL7 “Unified Codes for Units of Measure”

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>339</td>
<td>Acceleration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>342</td>
<td>Acceleration, angular</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept id</td>
<td>Rubric (en)</td>
<td>Description (en)</td>
<td>Mappings</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>------------------</td>
<td>----------</td>
</tr>
<tr>
<td>381</td>
<td>Amount (Eq)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>384</td>
<td>Amount (mole)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>497</td>
<td>Angle, plane</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Angle, solid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>335</td>
<td>Area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>Density</td>
<td></td>
<td></td>
</tr>
<tr>
<td>362</td>
<td>Diffusion coefficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>501</td>
<td>Electrical capacitance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>498</td>
<td>Electrical charge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>Electrical conductance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>334</td>
<td>Electrical current</td>
<td></td>
<td></td>
</tr>
<tr>
<td>377</td>
<td>Electrical field strength</td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>Energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>366</td>
<td>Energy density</td>
<td></td>
<td></td>
</tr>
<tr>
<td>508</td>
<td>Energy dose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>365</td>
<td>Energy per area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>347</td>
<td>Flow rate, mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>352</td>
<td>Flow rate, mass/force</td>
<td></td>
<td></td>
</tr>
<tr>
<td>351</td>
<td>Flow rate, mass/volume</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>Flow rate, volume</td>
<td></td>
<td></td>
</tr>
<tr>
<td>348</td>
<td>Flux, mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>355</td>
<td>Force</td>
<td></td>
<td></td>
</tr>
<tr>
<td>357</td>
<td>Force, body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>382</td>
<td>Frequency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>373</td>
<td>Heat transfer coefficient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>505</td>
<td>Illuminance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>379</td>
<td>Inductance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>499</td>
<td>Light intensity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>Loudness</td>
<td></td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>Luminous flux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>378</td>
<td>Magnetic flux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>Magnetic flux density</td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>Mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>385</td>
<td>Mass (IU)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concept id</td>
<td>Rubric (en)</td>
<td>Description (en)</td>
<td>Mappings</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>349</td>
<td>Mass per area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>344</td>
<td>Moment inertia, area</td>
<td></td>
<td></td>
</tr>
<tr>
<td>345</td>
<td>Moment inertia, mass</td>
<td></td>
<td></td>
</tr>
<tr>
<td>340</td>
<td>Momentum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>346</td>
<td>Momentum, flow rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>343</td>
<td>Momentum, angular</td>
<td></td>
<td></td>
</tr>
<tr>
<td>369</td>
<td>Power density</td>
<td></td>
<td></td>
</tr>
<tr>
<td>368</td>
<td>Power flux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>367</td>
<td>Power, linear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>Pressure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>507</td>
<td>Proportion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>380</td>
<td>Qualified real</td>
<td>This is a number with an arithmetic qualification (which may be no units, $10^3$ etc) allowing integers to be expressed as reals raised to a nominated power, or for real numbers alone.</td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>Radioactivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>375</td>
<td>Resistance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>370</td>
<td>Specific energy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>371</td>
<td>Specific heat, gas content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>337</td>
<td>Specific surface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>336</td>
<td>Specific volume</td>
<td></td>
<td></td>
</tr>
<tr>
<td>356</td>
<td>Surface tension</td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>Temperature</td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>338</td>
<td>Velocity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>341</td>
<td>Velocity, angular</td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>Velocity, dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>Velocity, kinematic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>374</td>
<td>Voltage, electrical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>Volume</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>Work</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.3.8 Null Flavours
This vocabulary codifies “flavours of null” for missing data items.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>271</td>
<td>“no information”</td>
<td>No information provided; nothing can be inferred as to the reason why, including whether there might be a possible applicable value or not.</td>
<td>HL7_NullFlavor::V10610</td>
</tr>
<tr>
<td>253</td>
<td>“unknown”</td>
<td>A possible value exists but is not provided.</td>
<td>HL7_NullFlavor::V10612</td>
</tr>
<tr>
<td>272</td>
<td>“masked”</td>
<td>The value has not been provided due to privacy settings.</td>
<td>HL7_NullFlavor::17932</td>
</tr>
<tr>
<td>273</td>
<td>“not applicable”</td>
<td>No valid value exists for this data item.</td>
<td>HL7_NullFlavor::10611</td>
</tr>
</tbody>
</table>

2.3.9 Participation Function
This vocabulary codifies functions of participation of parties in an interaction (used in PARTICIPATION class).

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>“not specified”</td>
<td>Mode of participation is not specified; use only for legacy data.</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>“face-to-face communication”</td>
<td>Face to face communications between parties in the same room.</td>
<td>HL7_ParticipationMode::16545</td>
</tr>
<tr>
<td>223</td>
<td>“interpreted face-to-face communication”</td>
<td>Face to face communications between parties in the same room with an interpreter</td>
<td>HL7_ParticipationMode::16545</td>
</tr>
<tr>
<td>217</td>
<td>“signing (face-to-face)”</td>
<td>Live face-to-face communication using a recognised sign language.</td>
<td></td>
</tr>
</tbody>
</table>

2.3.10 Participation Mode
This vocabulary codifies modes of participation of parties in an interaction (used in PARTICIPATION class). The initial set has been defined to be the same as HL7’s ParticipationMode vocabulary domain.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>“not specified”</td>
<td>Mode of participation is not specified; use only for legacy data.</td>
<td></td>
</tr>
<tr>
<td>216</td>
<td>“face-to-face communication”</td>
<td>Face to face communications between parties in the same room.</td>
<td>HL7_ParticipationMode::16545</td>
</tr>
<tr>
<td>223</td>
<td>“interpreted face-to-face communication”</td>
<td>Face to face communications between parties in the same room with an interpreter</td>
<td>HL7_ParticipationMode::16545</td>
</tr>
<tr>
<td>217</td>
<td>“signing (face-to-face)”</td>
<td>Live face-to-face communication using a recognised sign language.</td>
<td></td>
</tr>
<tr>
<td>Concept id</td>
<td>Rubric (en)</td>
<td>Description (en)</td>
<td>Mappings</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>195</td>
<td>“live audiovisual; videoconference; videophone”</td>
<td>Any audio-visual communication in real time</td>
<td></td>
</tr>
<tr>
<td>198</td>
<td>“videoconferencing”</td>
<td>Live audio-visual communication over video-conferencing or other similar equipment.</td>
<td>HL7_ParticipationMode::16548</td>
</tr>
<tr>
<td>197</td>
<td>“videophone”</td>
<td>Live audio-visual communication</td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>“signing over video”</td>
<td>Live video communication using sign language.</td>
<td></td>
</tr>
<tr>
<td>224</td>
<td>“interpreted video communication”</td>
<td>Live audio-visual communication involving an interpreter</td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>“asynchronous audio-visual; recorded video”</td>
<td>Audio-visual communication that is not live</td>
<td></td>
</tr>
<tr>
<td>196</td>
<td>“recorded video”</td>
<td>Recorded video or video mail</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>“live audio-only; telephone; internet phone; teleconference”</td>
<td>Any live audio-only communication.</td>
<td>HL7_ParticipationMode::V16544 (includes live)</td>
</tr>
<tr>
<td>204</td>
<td>“telephone”</td>
<td>Live verbal communication over a telephone.</td>
<td>HL7_ParticipationMode::16546</td>
</tr>
<tr>
<td>203</td>
<td>“teleconference”</td>
<td>Live verbal communication over teleconference</td>
<td>HL7_ParticipationMode::16546</td>
</tr>
<tr>
<td>204</td>
<td>“internet telephone”</td>
<td>Live verbal communication over the internet.</td>
<td>HL7_ParticipationMode::16546</td>
</tr>
<tr>
<td>222</td>
<td>“interpreted audio-only”</td>
<td>Any live audio-only communication using an interpreter.</td>
<td>HL7_ParticipationMode::V16544 (includes live)</td>
</tr>
<tr>
<td>199</td>
<td>“asynchronous audio-only; dictated; voice mail”</td>
<td>Audio-only communication that is not live.</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>“dictated”</td>
<td>Non-interactive audio-only information recorded on some medium, such as cassette tape.</td>
<td>HL7_ParticipationMode::16547</td>
</tr>
<tr>
<td>201</td>
<td>“voice-mail”</td>
<td>Audio messaging system</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>“live text-only; internet chat; SMS chat; interactive written note”</td>
<td>Any live text-only communication</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>“internet chat”</td>
<td>Live text-only communication over the internet</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>“SMS chat”</td>
<td>Live text-only chat over mobile/cell phone</td>
<td></td>
</tr>
<tr>
<td>215</td>
<td>“interactive written note”</td>
<td>Live text-only communication using written notes</td>
<td>HL7_ParticipationMode::16550</td>
</tr>
<tr>
<td>206</td>
<td>“asynchronous text; email; fax; letter; handwritten note; SMS message”</td>
<td>Any text-only communication including email, written text, SMS message etc.</td>
<td>HL7_ParticipationMode::V16549</td>
</tr>
<tr>
<td>211</td>
<td>“handwritten note”</td>
<td>Written communication by handwritten document.</td>
<td>HL7_ParticipationMode::16550</td>
</tr>
</tbody>
</table>
### 2.3.11 Related Party relationship

This vocabulary codifies the relationship between the subject of care and some other party mentioned in the health record.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en-uk)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>“self”</td>
<td>The party is the subject of EHR</td>
<td>HL7:RoleCode::</td>
</tr>
<tr>
<td>3</td>
<td>“foetus”</td>
<td>The party is a foetus</td>
<td>CEN:</td>
</tr>
<tr>
<td>10</td>
<td>“mother”</td>
<td>The party is the mother of the subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>9</td>
<td>“father”</td>
<td>The party is the father of the subject of the EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>6</td>
<td>“donor”</td>
<td>The party is a donor of organs or other body products to the EHR subject.</td>
<td>CEN:</td>
</tr>
<tr>
<td>253</td>
<td>“unknown”</td>
<td>Relationship to party is unknown.</td>
<td>CEN:</td>
</tr>
<tr>
<td>261</td>
<td>“adopted daughter”</td>
<td>Relationship of adopted daughter to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>260</td>
<td>“adopted son”</td>
<td>Relationship of adopted son to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>259</td>
<td>“adoptive father”</td>
<td>Relationship of adoptive father to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>258</td>
<td>“adoptive mother”</td>
<td>Relationship of adoptive mother to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>256</td>
<td>“biological father”</td>
<td>Relationship of biological father to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>255</td>
<td>“biological mother”</td>
<td>Relationship of biological mother to subject of EHR</td>
<td>CEN:</td>
</tr>
<tr>
<td>Concept id</td>
<td>Rubric (en-uk)</td>
<td>Description (en)</td>
<td>Mappings</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>-----------------</td>
<td>----------</td>
</tr>
<tr>
<td>23</td>
<td>“brother”</td>
<td>Relationship of brother to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>28</td>
<td>“child”</td>
<td>Relationship of child to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>265</td>
<td>“cohabitee”</td>
<td>Lives with the subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>257</td>
<td>“cousin”</td>
<td>Relationship of cousin to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>29</td>
<td>“daughter”</td>
<td>Relationship of daughter to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>264</td>
<td>“guardian”</td>
<td>Relationship of guardian to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>39</td>
<td>“maternal aunt”</td>
<td>Relationship of maternal aunt to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>8</td>
<td>“maternal grandfather”</td>
<td>Relationship of maternal grandfather to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>7</td>
<td>“maternal grandmother”</td>
<td>Relationship of maternal grandmother to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>38</td>
<td>“maternal uncle”</td>
<td>Relationship of maternal uncle to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>189</td>
<td>“neonate”</td>
<td>Relationship of neonate to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>254</td>
<td>“parent”</td>
<td>Relationship of parent to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>22</td>
<td>“partner/spouse”</td>
<td>The husband or wife or life partner of the subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>41</td>
<td>“paternal aunt”</td>
<td>Relationship of paternal aunt to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>36</td>
<td>“paternal grandfather”</td>
<td>Relationship of paternal grandfather to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>37</td>
<td>“paternal grandmother”</td>
<td>Relationship of paternal grandmother to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>40</td>
<td>“paternal uncle”</td>
<td>Relationship of paternal uncle to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>27</td>
<td>“sibling”</td>
<td>Relationship of sibling to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>24</td>
<td>“sister”</td>
<td>Relationship of sister to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>31</td>
<td>“son”</td>
<td>Relationship of son to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>263</td>
<td>“step father”</td>
<td>Relationship of step father to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>262</td>
<td>“step mother”</td>
<td>Relationship of step mother to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>25</td>
<td>“step or half brother”</td>
<td>Relationship of step or half brother to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
<tr>
<td>26</td>
<td>“step or half sister”</td>
<td>Relationship of step or half sister to subject of EHR</td>
<td>HL7: CEN:</td>
</tr>
</tbody>
</table>
2.3.12 Setting
This vocabulary codifies broad types of settings in which clinical care is delivered. It is not intended to be a perfect classification of the real world, but instead a practical coarse-grained categorisation to aid querying.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>“home”</td>
<td>Care delivered in the patient’s home by patient or health professional.</td>
</tr>
<tr>
<td>227</td>
<td>“emergency care”</td>
<td>Care delivered in emergency situation, e.g. by ambulance workers.</td>
</tr>
<tr>
<td>228</td>
<td>“primary medical care”</td>
<td>Care delivered by a doctor within a primary care framework (generalist, non-referred).</td>
</tr>
<tr>
<td>229</td>
<td>“primary nursing care”</td>
<td>Care delivered by nurses within a primary care framework (community based, generalist clinic).</td>
</tr>
<tr>
<td>230</td>
<td>“primary allied health care”</td>
<td>Care delivered by allied health practitioners such as physiotherapists, osteopaths, chiropractors, optometrists, chiropodist/pediatrist etc. within a primary care framework (community based, generalist clinic).</td>
</tr>
<tr>
<td>231</td>
<td>“midwifery care”</td>
<td>Midwifery care in any framework</td>
</tr>
<tr>
<td>232</td>
<td>“secondary medical care”</td>
<td>Care delivered in an institutional or specialist setting - usually as a result of a referral.</td>
</tr>
<tr>
<td>233</td>
<td>“secondary nursing care”</td>
<td>Care delivered by nurses within a secondary care framework (inpatient, specialist clinic).</td>
</tr>
<tr>
<td>234</td>
<td>“secondary allied health care”</td>
<td>Care delivered by allied health care professionals within a secondary care framework (inpatient, specialist clinic).</td>
</tr>
<tr>
<td>235</td>
<td>“complementary health care”</td>
<td>Care delivered by chinese, ayurvedic, naturopath, homeopath etc practitioner.</td>
</tr>
<tr>
<td>236</td>
<td>“dental care”</td>
<td>Care delivered in a dental practitioner setting.</td>
</tr>
<tr>
<td>237</td>
<td>“nursing home care”</td>
<td>Care to the needs of patients in nursing homes, delivered in an institutional setting.</td>
</tr>
<tr>
<td>238</td>
<td>“other care”</td>
<td>Care delivered in setting not described by other terms in this vocabulary.</td>
</tr>
</tbody>
</table>

2.3.13 Term Mapping Purpose
This vocabulary codifies purposes for term mappings as used in the class TERM_MAPPING. The use-case for this vocabulary is yet to be determined.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2.3.14 Version Lifecycle State
This vocabulary codifies lifecycle states of Compositions or other elements of the health record.

<table>
<thead>
<tr>
<th>Concept id</th>
<th>Rubric (en)</th>
<th>Description (en)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>532</td>
<td>“complete”</td>
<td>Item is complete at time of committal.</td>
<td></td>
</tr>
<tr>
<td>553</td>
<td>“incomplete”</td>
<td>Item is incomplete at time of committal, in the view of the author. Further editing or review needed before its status will be set to “finished”.</td>
<td></td>
</tr>
<tr>
<td>523</td>
<td>“deleted”</td>
<td>Item has been logically deleted.</td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>“draft”</td>
<td>Item is in draft state: not ready for viewing by other users. DEPRECATED.</td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>“active”</td>
<td>Item is active and available for shared use. DEPRECATED.</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>“inactive”</td>
<td>Item is marked inactive due to logical deletion or other similar operation. DEPRECATED.</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>“awaiting approval”</td>
<td>Item is awaiting to approval to go into active state. DEPRECATED</td>
<td>HL7_ParticipationSignature::10283</td>
</tr>
</tbody>
</table>
REFERENCE MODEL

The openEHR EHR Information Model

Editors: {T Beale, S Heard}¹, {D Kalra, D Lloyd}²

Revision: 5.0
Pages: 83

1. Ocean Informatics Australia
2. Centre for Health Informatics and Multi-professional Education, University College London

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
</table>

### R E L E A S E  0.96

### R E L E A S E  0.95


### R E L E A S E  0.9

Editors:{T Beale, S Heard}, {D Kalra, D Lloyd} Page 3 of 83 Date of Issue: 25 Jan 2006

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>CR-000096. Allow 0..* SECTIONs as COMPOSITION content.</td>
<td>DSTC</td>
<td>11 Mar 2004</td>
</tr>
</tbody>
</table>
| 4.4     | **CR-000019.** Add HISTORY & STRUCTURE supertype. **CR-000028.** Change name of STRUCTURE class to avoid clashes.  
|         | **CR-000087.** EVENT_CONTEXT.location should be optional.  
|         | **CR-000088.** Move INSTRUCTION guideline_id to ENTRY.  
|         | **CR-000092.** Improve EVENT_CONTEXT modelling. Rename author to composer.  
|         | Formally validated using ISE Eiffel 5.4.                              | T Beale, H Frankel, DSTC    | 06 Mar 2004 |
| 4.3.10  | CR-000044. Add reverse ref from VERSION_REPOSITORY<T> to owner object. Add invariants to DIRECTORY and VERSIONED_COMPOSITION classes.  
|         | **CR-000046.** Rename COORDINATED_TERM and DV_CODED_TEXT.definition.  
|         |                                                                  | T Beale                     | 25 Feb 2004 |
| 4.3.9   | CR-000021. Rename CLINICAL_CONTEXT.practice_setting to setting.        | A Goodchild                 | 10 Feb 2004 |
| 4.3.8   | CR-000057. Environmental information needs to be included in the EHR.  | T Beale                     | 02 Nov 2003 |
| 4.3.7   | **CR-000048.** Pre-release review of documents.  
|         | **CR-000049.** Correct reference types in EHR, DIRECTORY classes. EHR.contributions, all_compositions, FOLDER.compositions attributes and invariants corrected.  
|         | **CR-000050.** Update Path syntax reference model to ADL specification.  
| 4.3.6   | **CR-000041.** Visually differentiate primitive types in openEHR documents.  
|         |                                                                  | D Lloyd                     | 04 Oct 2003 |
| 4.3.5   | CR-000013. Rename key classes, according to CEN ENV 13606.              | S Heard, D Kalra, T Beale   | 15 Sep 2003 |
| 4.3.4   | **CR-000011.** Add author attribute to EVENT_CONTEXT.  
|         | **CR-000027.** Move feeder_audit to LOCATABLE to be compatible with CEN 13606 revision.  
|         |                                                                  | S Heard, D Kalra            | 20 Jun 2003 |
| 4.3.3   | **CR-000020.** Move VERSION.territory to TRANSACTION.  
|         | **CR-000018.** Add DIRECTORY class to RM.EHR Package.  
|         | **CR-000005.** Rename CLINICAL_CONTEXT to EVENT_CONTEXT.  
|         |                                                                  | A Goodchild                 | 10 Jun 2003 |
| 4.3.2   | **CR-000006.** Make ENTRY.provider a PARTICIPATION.  
|         | **CR-000007.** Replace ENTRY.subject and subject_relationship with RELATED_PARTY.  
|         | **CR-000008.** Remove confidence and is_exceptional attributes from ENTRY.  
|         | **CR-000009.** Merge ENTRY protocol and reasoning attributes.  
<p>|         |                                                                  | S Heard, T Beale, D Kalra, D Lloyd | 11 Apr 2003 |
| 4.3.1   | DSTC review - typos corrected.                                         | A Goodchild                 | 08 Apr 2003 |</p>
<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Formally validated using ISE Eiffel 5.2. Moved VERSIONED_TRANSACTION class to EHR Package, to correspond better with serialised formalisms like XML.</td>
<td>T Beale, A Goodchild</td>
<td>25 Feb 2003</td>
</tr>
<tr>
<td>4.1</td>
<td>Changes post CEN WG meeting Rome Feb 2003. Moved TRANSACTION.version_id postcondition to an invariant. Moved feeder_audit back to TRANSACTION. Added ENTRY.act_id, VERSION_AUDIT.attestations moved to new ATTESTATIONS class attached to VERSIONED&lt;T&gt;.</td>
<td>T Beale, S Heard, D Kalra, D Lloyd</td>
<td>8 Feb 2003</td>
</tr>
<tr>
<td>4.0.2</td>
<td>Various corrections and DSTC change requests. Reverted OBSERVATION.items:LIST&lt;HISTORY&lt;T&gt;&gt; to data:HISTORY&lt;T&gt; and EVALUATION.items:LIST&lt;STRUCTURE&lt;T&gt;&gt; to data:STRUCTURE&lt;T&gt;. Changed CLINICAL_CONTEXT.other_context to a STRUCTURE. Added ENTRY.other_participations; Added CLINICAL_CONTEXT.participations; removed hcp_legally_responsible (to be archetyped). Replaced EVENT_TRANSACTION and PERSISTENT_TRANSACTION with TRANSACTION and a boolean attribute is_persistent.</td>
<td>T Beale</td>
<td>3 Feb 2003</td>
</tr>
<tr>
<td>4.0.1</td>
<td>Detailed corrections to diagrams and class text from DSTC.</td>
<td>Z Tun</td>
<td>8 Jan 2003</td>
</tr>
<tr>
<td>4.0</td>
<td>Moved HISTORY classes to Data Structures RM. No semantic changes.</td>
<td>T Beale</td>
<td>18 Dec 2002</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Corrections on 3.8.1. No semantic changes.</td>
<td>D Lloyd</td>
<td>11 Nov 2002</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Removed SUB_FOLDER class. Now folder structure can be nested separately archetyped folder structures, same as for ORGANISERS. Removed AUTHORED_TA and ACQUISITION_TA classes; simplified versioning.</td>
<td>T Beale, D Kalra, D Lloyd, A Goodchild</td>
<td>28 Oct 2002</td>
</tr>
<tr>
<td>3.8</td>
<td>Added practice_setting attribute to CLINICAL_CONTEXT, inspired from HL7v3/ANSI CDA standard Release 2.0. Changed DV.PLAIN_TEXT to DV.TEXT. Removed hca_coauthorising; renamed hca_recording; adjusted all instances of *_ID; converted CLINICAL_CONTEXT.start_time, end_time to an interval.</td>
<td>T Beale, S Heard, D Kalra, M Darlison</td>
<td>22 Oct 2002</td>
</tr>
<tr>
<td>3.7</td>
<td>Removed Spatial package to Common RM document. Renamed ACTION back to ACTION_SPECIFICATION. Removed the class NAVIGABLE_STRUCTURE. Renamed SPATIAL to STRUCTURE. Removed classes STATE_HISTORY, STATE, SINGLE_STATE. Removed Communication (EHR_EXTRACT) section to ow document.</td>
<td>T Beale</td>
<td>22 Sep 2002</td>
</tr>
<tr>
<td>3.6</td>
<td>Removed Common and Demographic packages to their own documents.</td>
<td>T Beale</td>
<td>28 Aug 2002</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Altered syntax of EXTERNAL_ID identifiers.</td>
<td>T Beale, Z Tun</td>
<td>20 Aug 2002</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>3.5</td>
<td>Rewrote Demographic and Ehr_extract packages.</td>
<td>T Beale</td>
<td>18 Aug 2002</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Simplified EHR_EXTRACT model, numerous small changes from DSTC review.</td>
<td>T Beale, Z Tun</td>
<td>15 Aug 2002</td>
</tr>
<tr>
<td>3.3</td>
<td>Rewrite of contributions, version control semantics.</td>
<td>T Beale, D Lloyd, D Kalra, S Heard</td>
<td>1 Aug 2002</td>
</tr>
<tr>
<td>3.2</td>
<td>DSTC comments. Various minor errors/omissions. Changed inheritance of SINGLE_EVENT and SINGLE_STATE. Included STRUCTURE subtype methods from GEHR. ehr_id added to VT. Altered EHR/FOLDER attrs. Added EXTERNAL_ID.version.</td>
<td>T Beale, Z Tun</td>
<td>25 Jun 2002</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Minor corrections.</td>
<td>T Beale</td>
<td>20 May 2002</td>
</tr>
<tr>
<td>3.1</td>
<td>Reworking of Structure section, Action class, Instruction class.</td>
<td>T Beale, S Heard</td>
<td>16 May 2002</td>
</tr>
<tr>
<td>3.0</td>
<td>Plans, actions updated.</td>
<td>T Beale, S Heard</td>
<td>10 May 2002</td>
</tr>
<tr>
<td>2.9</td>
<td>Additions from HL7v3 coded term model, alterations to quantity model, added explanation sections.</td>
<td>T Beale</td>
<td>5 May 2002</td>
</tr>
<tr>
<td>2.8.2a</td>
<td>Interim version with various review modifications</td>
<td>T Beale</td>
<td>28 Apr 2002</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Error corrections to EHR_EXTRACT package. P Schloeffel comments on 2.7.</td>
<td>T Beale, P Schloeffel</td>
<td>25 Apr 2002</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Further minor changes from UCL on v2.7.</td>
<td>T Beale</td>
<td>24 Apr 2002</td>
</tr>
<tr>
<td>2.8</td>
<td>Dipak Kalra (UCL) comments on v2.6 incorporated. Added External Package. Minor changes elsewhere.</td>
<td>T Beale, D Kalra</td>
<td>23 Apr 2002</td>
</tr>
<tr>
<td>2.7</td>
<td>Final development of initial draft, including EHR_EXTRACT, related models</td>
<td>T Beale</td>
<td>20 Apr 2002</td>
</tr>
<tr>
<td>2.6</td>
<td>Further development of path syntax, incorporation of Dipak Kalra’s comments</td>
<td>T Beale, D Kalra</td>
<td>15 Apr 2002</td>
</tr>
<tr>
<td>2.5</td>
<td>Further development of clinical and record management clusters.</td>
<td>T Beale</td>
<td>10 Apr 2002</td>
</tr>
<tr>
<td>2.4</td>
<td>Included David Lloyd’s rev 2.3 comments.</td>
<td>T Beale, D Lloyd</td>
<td>4 Apr 2002</td>
</tr>
<tr>
<td>2.3</td>
<td>Improved context analysis.</td>
<td>T Beale</td>
<td>4 Mar 2002</td>
</tr>
<tr>
<td>2.2</td>
<td>Added path syntax.</td>
<td>T Beale</td>
<td>19 Nov 2001</td>
</tr>
<tr>
<td>2.1</td>
<td>Minor organisational changes, some content additions.</td>
<td>T Beale</td>
<td>18 Nov 2001</td>
</tr>
<tr>
<td>1.2</td>
<td>Major additions to introduction, design philosophy</td>
<td>T Beale</td>
<td>1 Nov 2001</td>
</tr>
<tr>
<td>1.1</td>
<td>Major changes to diagrams; STILL UNREVIEWED</td>
<td>T Beale</td>
<td>13 Oct 2001</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------</td>
<td>---------</td>
<td>---------------</td>
</tr>
<tr>
<td>1.0</td>
<td>Based on GEHR Object Model</td>
<td>T Beale</td>
<td>22 Sep 2001</td>
</tr>
</tbody>
</table>
Acknowledgements

Thanks to...

The work reported in this paper has been funded in by a number of organisations, including University College, London; The Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia; Ocean Informatics, Australia.

Andrew Goodchild, senior research scientist at the Distributed Systems Technology Centre, Brisbane provided valuable in-depth comments and insights on all aspects of the model during its early development.

CORBA is a trademark of the Object Management Group

.Net is a trademark of Microsoft Corporation
1 Introduction .................................................................................. 12
1.1 Purpose ...................................................................................... 12
1.2 Related Documents .................................................................... 12
1.3 Status ....................................................................................... 12
1.4 Peer review ................................................................................ 12
1.5 Conformance ............................................................................. 13

2 Background .................................................................................. 14
2.1 Requirements ........................................................................... 14
2.1.1 Original GEHR Requirements ........................................... 14
2.1.2 GEHR Australia Requirements ......................................... 14
2.1.3 European Synapses and SynEx Project Requirements ....... 14
2.1.4 European EHCR Support Action Requirements ............... 15
2.1.5 ISO EHR Requirements ..................................................... 15
2.1.6 openEHR Requirements ..................................................... 15
2.2 Design Principles ..................................................................... 15
2.2.1 The System-of-systems EHR Context ................................ 16
2.2.2 Context Model of Recording ............................................. 16
2.3 Relationship to other Health Information Models .................. 17
2.3.1 CEN TC/251 prEN13606 ...................................................... 17
2.3.2 HL7 Version 3 ..................................................................... 18
2.3.3 OMG HDTF ....................................................................... 18

3 The EHR Information Model ..................................................... 19
3.1 Overview ................................................................................... 19

4 The EHR ..................................................................................... 21
4.1 Overview ................................................................................... 21
4.2 Design Principles ..................................................................... 21
4.2.1 Compositions ...................................................................... 21
4.2.2 Folders ............................................................................... 23
4.2.3 EHR Status ........................................................................ 24
4.2.4 Change Control of the EHR .............................................. 25
4.2.5 Versioning of Compositions .............................................. 26
4.2.6 Versioning Scenarios ......................................................... 27

5 EHR Package ............................................................................. 29
5.1 Overview ................................................................................... 29
5.2 EHR Creation Semantics ....................................................... 30
5.3 Time in the EHR ..................................................................... 30
5.4 Historical Views of the Record .............................................. 31
5.5 Class Descriptions ................................................................. 31
5.5.1 EHR Class ........................................................................ 31
5.5.2 VERSIONED_EHR_STATUS Class .................................. 32
5.5.3 EHR_STATUS Class ......................................................... 33
5.5.4 VERSIONED_COMPOSITION Class ............................... 33

6 Composition Package ............................................................. 35
6.1 Overview ................................................................................... 35
6.2 Design Principles ................................................................... 35
6.2.1 Composition Composer .................................................... 35
| B.1  | General ........................................................................................................... | 79 |
| B.2  | European Projects .................................................................................... | 80 |
| B.3  | CEN ............................................................................................................ | 80 |
| B.4  | GEHR Australia ....................................................................................... | 81 |
| B.5  | HL7v3 ........................................................................................................ | 81 |
| B.6  | OMG .......................................................................................................... | 81 |
| B.7  | Software Engineering .............................................................................. | 81 |
| B.8  | Resources ............................................................................................... | 82 |
Introduction

1.1 Purpose
This document describes the openEHR EHR Information Model, which is a model of an interoperable EHR in the ISO RM/ODP information viewpoint. This model defines a logical EHR information architecture rather than just an architecture for communication of EHR extracts or documents between EHR systems. The openEHR definition of the EHR Extract is given in the openEHR EHR_EXTRACT Information Model.

The intended audience includes:

- Standards bodies producing health informatics standards
- Software development groups using openEHR
- Academic groups using openEHR
- The open source healthcare community

1.2 Related Documents
Prerequisite documents for reading this document include:

- The openEHR Modelling Guide
- The openEHR Support Information Model
- The openEHR Data Types Information Model
- The openEHR Common Information Model

Other documents describing related models, include:

- The openEHR EHR_EXTRACT Information Model
- The openEHR Demographic Information Model

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

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

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

Blue text indicates sections under active development.

1.4 Peer review
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.

1.5 Conformance

Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Background

This section describes the inputs to the modelling process that created the openEHR Information Model.

2.1 Requirements

There are broadly three sets of requirements which inform this model, as described in the following subsections.

2.1.1 Original GEHR Requirements

From the European GEHR project (1992-5; [19]), the following broad requirements areas were identified:

- the life-long EHR;
- priority: Clinician / Patient interaction;
- medico-legal faithfulness, traceability, audit-trailing;
- technology & data format independent;
- facilitate sharing of EHRs;
- suitable for both primary & acute care;
- secondary uses: education, research, population medicine;
- open standard & software deliverables;

The original deliverables can be reviewed in detail at the GEHR page at CHIME, UCL.

2.1.2 GEHR Australia Requirements

The GEHR Australia project (1997-2001; [32], [33]) introduced further requirements, including:

- support for clinical data structures: lists, tables, time-series etc;
- safer information model than the original (European) GEHR: context attributes only in valid places (but still similar style);
- separate groups for “persistent”, “demographic” and “event” information in EHR, which corresponds closely to real clinical querying patterns;
- introduction of formally specified archetypes, and archetype-enabled information model;
- interoperability at the knowledge level, i.e. level of domain definitions of information such as “discharge summary” and “biochemistry result”;
- XML-enabled;
- consider compatibility with CEN 13606, Corbamed, HL7v3.

GEHR Australia produced a proof of concept implementation in which clinical archetypes were developed and used. See [3] for a technical description of archetypes.

2.1.3 European Synapses and SynEx Project Requirements

Following the original Good European Health Record project the EU-funded Synapses (1996-8; [25]) and SynEx (1998-2000; [14]) projects extended the original requirements basis of GEHR to include further requirements, as follows:

- the requirements of a federation approach to unifying disparate clinical databases and EPR systems: the federated health record (FHR) [26];
• the need to separate a generic and domain independent high-level model for the federated health record from the (closely related) model of the meta-data which defines the domain specific health record characteristics of any given clinical specialty and any given federation of database schemata;
• a formalism to define and communicate (share) knowledge about the semantic hierarchical organisation of an FHR, the permitted data values associated with each leaf node in a record hierarchy and any constraints on values that leaf nodes may take (the Synapses Object Dictionary) [27];
• the core technical requirements of and interfaces for a federation middleware service [25].

2.1.4 European EHCR Support Action Requirements
This EU Support Action project (“SupA”; [17], [18], [20]) consolidated the requirements published by a wide range of European projects and national health informatics organisations as a Consolidated List of Requirements [16].

2.1.5 ISO EHR Requirements
The above requirements publications and the recent experience of openEHR feed into the definition of a set of EHR requirements authored by ISO Technical Committee 215 (Health Informatics) - ISO TS 18308. The present draft [13] has been reviewed by the authors of this document and openEHR will seek to maintain a close mapping between its information models and services and this international requirements work. The openEHR mapping to ISO 18308 can be found on the openEHR website.

2.1.6 openEHR Requirements
New requirements generated during the development of openEHR included the following:
• major improvements in information models from the point of view of reducing programmer errors and ambiguity;
• better modelling of time and context (temporal/spatial approach);
• better understanding of legacy system / federation problem;
• workflow modelling;
• harmonise with CEN EN 13606;
• integration with HL7v2 and other messaging systems;
• numerous specific clinical requirements.

2.2 Design Principles
There are various considerations outside the requirements which influence the EHR information model, some of which are described by Beale & Heard, e.g. [4], [5], [6]. These are summarised as follows:
• system-of-systems understanding of information infrastructures (i.e. a set of collaborating middleware components and services), described in more detail below;
• ontological framework within which reference models and archetypes fit;
• design paradigm: two level modelling, archetypes;
• separation of models according to responsibilities;
2.2.1 The System-of-systems EHR Context

Health care information systems, both at the level of a single enterprise and regional healthcare networks, will increasingly comprise a set of software components collaborating through middleware and web services, enabling distributed operations and data exchange. This “system-of-systems” view embraces the distributed object paradigm (exemplified by CORBA and .Net), the message-based paradigm (exemplified by HL7), and in a less formal sense, most of today’s organically evolved multi-database hospital IT environments. The distributed systems paradigm is also the generally accepted theoretical and standards view, as exemplified by any modern textbook on information processing, by the ISO RM/ODP standard, by health informatics standards such as OMG’s Corbamed [37], [38], [39] specifications and CEN family of health information standards ([28] and many others), and by a wide range of international projects, national health information strategies and demonstrator pilots.

Accordingly, the EHR is understood in openEHR as one system (or service) within a distributed health information infrastructure, whose purpose is to manage the longitudinal and comprehensive EHR of individual patients. There is no definitive list of services or systems required within a health care environment. However, it is generally understood that in any deployment scenario an EHR service would be complemented by a range of other services including: terminology, clinical reference data (prescribing, interactions), order management, scheduling, decision support, demographics (both patient and health provider), pathology, imaging, and access control. In a distributed system infrastructure, each of these will usually exist as a service within an infrastructure node. Each system has its own information model, describing the semantics of the data which can be obtained from it or written to it, as well as its service interface describing the functional interface to the system.

The approach taken in defining the openEHR EHR Information Model has been to assume this distributed deployment environment, and therefore the existence of several other services, at minimum for demographics, access control, terminology and archetypes. The Information Model therefore contains classes and attributes to facilitate interoperability with such services rather than their duplication inside the EHR service.

However, all service-level information models in openEHR - that is to say, the information models of the EHR, demographics etc - share the same platform of support information models, consisting of Support IM, Data structures IM, Data types IM, and Common IM.

2.2.2 Context Model of Recording

The openEHR EHR model takes account of a systematic analysis of “context”, described in [5]. Contexts in the real world are mapped to particular levels of the information model in a clear way, according to the scheme shown in Figure 1. On the left hand side is depicted the context of a data-entry session in which the information generated by a “healthcare event”, containing “clinical statements”, is added to the EHR. A healthcare event is defined as any business activity of the health care system for the patient, including encounters, pathology tests, and interventions. A clinical statement is the minimal indivisible unit of information the clinician wants to record. Clinical statements are shown in the diagram having temporal and spatial structure as well as data values. Each of these three contexts has its own audit information, consisting of who, when, where, why information.
On the right hand side of FIGURE 1, the EHR recording environment is represented. The EHR consists of distinct, coarse-grained items known as Compositions added over time and organised by Folders. Each Composition consists of Entries, organised by Sections within the Composition. The audit information for each context is recorded at the corresponding level of the EHR.

### 2.3 Relationship to other Health Information Models

Where relevant, the correspondences to other information models have been documented in this specification. Correspondences to the GEHR Australia and EU Synapses/SynEx models are shown, since these are the models on which the openEHR EHR information model is primarily based. The following sections summarise the other models and standards with which correspondences are shown.

#### 2.3.1 CEN TC/251 prEN13606

These models have been influenced by and have also influenced the models in CEN prEN13606 (2005 revision); accordingly, relationships to 13606 have been documented fairly precisely.

Since January 2002, the prEN13606 prestandard has been the subject of significant revision, as part of its transition to a full European Standard (“EN”). This work has been influenced by the openEHR specifications, and has itself been a source of further insights and changes to the openEHR specifications. Particular areas of openEHR which have been changed due to this process include:

- change of major class names (TRANSACTION -> COMPOSITION etc; see CR-000013);
- improved model of ATTESTATION (see CR-000025);
- improved model of feeder audits (see CR-000027).

Implementation experience with Release 0.9 and 0.95 of openEHR has further improved these areas significantly. Nevertheless, openEHR is not a copy of CEN, for two reasons. Firstly, its scope includes systems, while EN13606 defines an EHR Extract; secondly, EN13606 suffers somewhat from “design by committee”, and has no formal validation mechanism for its models.
2.3.2  HL7 Version 3

Correspondences to some parts of HL7 version 3 (ballot 5, July 2003) are also documented where possible, however, it should be understood that there are a number of difficulties with this. Firstly, while the HL7v3 Reference Information Model (RIM) - the closest HL7 artifact to an information model - provides similar data types and some related semantics, it is not intended to be a model of the EHR. In fact, it differs from the information model presented here (and for that matter most published information models) in two basic respects: a) it is an amalgam of semantics from many systems which would exist in a distributed health information environment, rather than a model of just one (the EHR); b) it is also not a model of data, but an “analysis pattern” in the sense of Fowler [43] from which further specific models - subschemas - are developed by a custom process of “refinement by restriction”, in order to arrive at message definitions. As a consequence, data in messages are not instances of HL7v3 RIM classes, as would be the case in other systems based on information models of the kind presented here.

Despite the differences, there are some areas that appear to be candidates for fairly direct mapping, specifically the data types and terminology use, and the correspondence between openEHR Compositions and some parts of the HL7 Clinical Document Architecture (CDA).

2.3.3  OMG HDTF

In general, the openEHR information models represent a more recent analysis of the required semantics of EHR and related information than can be found in the information viewpoint of the OMG HDTF specifications (particularly PIDS and COAS). However, the computational viewpoint (i.e. functional service interface definitions) is one of the inputs to the openEHR service model development activity.
3 The EHR Information Model

3.1 Overview

FIGURE 2 illustrates the package structure of the openEHR EHR information model.

The packages are as follows:

ehr: This package contains the top level structure, the EHR, which consists of an EHR_STATUS object, a hierarchical directory of FOLDERs, containing references to VERSIONED_COMPOSITIONs, and a collection of CONTRIBUTIONS which document the changes to the EHR over time.

composition: The Composition is the EHR’s top level “data container”, and is described by the COMPOSITION class.

content: This package contains the Navigation and Entry packages, whose classes describe the structure and semantics of the contents of Compositions in the health record.

navigation: The SECTION class provides a navigational structure to the record, similar to “headings” in the paper record. ENTRYs and other SECTIONs can appear under SECTIONs.

entry: This package contains the generic structures for recording clinical statements. Entry types include ADMIN_ENTRY, OBSERVATION (for all observed phenomena, including mechanically or manually measured, and responses in interview), EVALUATION (for assessments, diagnoses, plans), INSTRUCTION (actionable statements such as medication orders, recalls, monitoring, reviews), and ACTION (information recorded as a result of performing Instructions).

FIGURE 3 illustrates an overview of the class structure of the EHR Information Model, along with the main concepts on which they rely, namely Data Types, Data Structures, Archetyped, and Identification.
FIGURE 3 openEHR and EHR Extract Information Model Overview
4 The EHR

4.1 Overview

The design of the openEHR EHR is based on concepts and experience from the GeHR Australia project [32], the Synapses project [25], [26], [27], the CEN prEN13606 standard (2005), and the HL7 Clinical Document Architecture (CDA) Release 1 [35]. The contents of an openEHR EHR consist of an EHR status object, Composition objects (equivalent to the CEN 13606 Composition and the HL7 CDA Document), and an optional directory of Folders (a CEN 13606 concept). All content in the EHR is versioned, by means of the VERSIONED_OBJECT<T> type (defined in the Common IM). This means that every previous state of the EHR - i.e. all states of the EHR status, the Compositions and the directory structure - is available. The semantics of versioning are provided by the concepts defined in the rm.common.change_control package. All changes to the EHR are made by committing Contributions (a similar idea to “change-sets” in configuration management (CM) systems), where a Contribution can include changes to the EHR status objects, creation, update or correction of Compositions, creation or modification of the Folder structure, move of Compositions in the Folder structure, and deletion of Compositions or part of the Folder structure. The use of Contributions guarantees that the EHR progresses from one valid state to another, regardless of what changes occur in any particular contribution.

The discussion below commences by describing the general organisation of the EHR, and then how change control applies to the EHR. It then describes the semantics of Compositions - the containers of data in the EHR, and finally describes the role of the Folder structure.

4.2 Design Principles

4.2.1 Compositions

The Composition concept in the openEHR EHR originated from the Transaction concept of the GEHR project [21], [22], [23], [24], which was based on the concept of a unit of information corresponding to the interaction of a healthcare agent with the EHR. It was originally designed to satisfy the following needs (which include the well-known ACID characteristics of transactions [8]):

- **durability**: the need for a persistent unit of information committal in the record;
- **atomicity**: the need for a minimal unit of integrity for clinical information, corresponding to a minimal unit for committal, transmission and security;
- **consistency**: the need for contributions to the record to leave the record in a consistent state;
- **isolation**: the need for contributions to the record by simultaneous users not to interfere with each other;
- **indelibility**: the requirement that information committed to the record be indelible in order to support later investigations, for both medico-legal and process improvement purposes, and the consequent requirement to be able to access previous states of the record;
- **modification**: the need for users to be able to modify EHR contents, in order to correct errors or update previously recorded information (e.g. current medications, family history); and
- **traceability**: the need to record adequate auditing information at committal, in order to provide clinical and legal traceability.

The Transaction concept has since been renamed to “Composition”, which is the name of the equivalent concept in the current CEN EN13606, and it has been expanded and more formally defined in
openEHR in two ways. Firstly, the idea of a unit of committal has been formalised by the openEHR model of change control (see the openEHR Common Information Model); how this applies to the EHR and compositions is described below. Secondly, the informational purpose of a Composition is no longer just to contain data from a passing clinical event such as a patient contact, but also to capture particular categories of clinical data which have long-lived significance, such as problem and medication lists.

Experience with health information systems, including the GEHR (Australia) project, SynEx, Synapses, and inspection of common commercial systems, has shown that there are two general categories of information at the coarse level which exist in the EHR: event items, and longitudinal, or persistent items, of which there might be various kinds.

**Event Compositions**

Events record what happens during the clinical session context [5] which occur due to billable healthcare system events with or for the patient, such as patient contacts, but also sessions in which the patient is not a participant (e.g. surgery) or not present (e.g. pathology testing). Persistent items capture information which remains valid in the long term, such as the patient’s family history, current medications, care plan and so on. Both types of information are contained within Compositions. FIGURE 4 illustrates a simple EHR comprising an accumulation of event Compositions.

![FIGURE 4 Basic Event-oriented EHR](image)

An important job of the event Composition is to record not only the data from the healthcare event, such as observations on the patient, but also to record the event context information, i.e. the who, when, where and why of the event. For this reason, a specific class representing clinical context is associated with event compositions in the formal model.

**Persistent Compositions**

However, in a more sophisticated EHR, there are also likely to be persistent compositions. Many items of long-term interest in the record are separated by clinicians into well-known categories, such as:

- Family history
- Social history
- Problem list
- Current medications
- Therapeutic precautions
- Vaccination history
- Lifestyle
- Care plan

Note that over time, the number of event Compositions is likely to far outstrip the number of persistent Compositions. FIGURE 5 illustrates an EHR containing persistent information as well as event information.

In any clinical session, an event composition will be created, and in many cases, persistent compositions will be modified. How this works is described below under Change Control of the EHR on page 25.
4.2.2 Folders

As Compositions accumulate over time in the EHR, they form a long-term patient history. Previous work in CEN [28] - [31] and SynEx [26] suggests that it is useful to be able to organise Compositions using a hierarchy of folders to classify them, much the same way files in a file system are arranged in the directory structure, as visualised by the Windows Explorer and other similar tools. In the openEHR model, folders do not contain Compositions by value but by reference, and are completely optional. More than one Folder can refer to the same Composition. Folders might be used to manage a simple classification of Compositions, e.g into event and persistent, or they might be used to create numerous categories, based on episodes or other groupings of Compositions. Folder structures can be archetyped.

A simple structure showing Folders referencing Compositions is shown in FIGURE 6, in which the following folders are used:

Subject: a composition containing clinically relevant demographic data of the patient;
Persistent: compositions containing information which is valid in the long term;
Event: compositions containing information whose currency is limited to the short term after the time of committal;

FIGURE 6 Using Folders to Group Compositions
Episode xxx: rather than using a single ‘event’ folder, it may be convenient to group event compositions into episodes (periods of treatment at a health care facility relating to one or more identified problems) and/or other categories such as on the basis of type of healthcare (orthodox, homeopathic, etc).

A justification for these particular categories is based on patterns of access. The persistent category consists of a dozen or so compositions described above, and which are continually required by querying (particularly lifestyle, current problems and medications). The event category consists of clinical data whose relevancy fades fairly quickly, including most measurements made on the patients or in pathology. Compositions in this category are thus potentially very numerous over the patient’s lifetime, but of decreasing relevance to the clinical care of the patient in time; it therefore makes sense to separate them from the persistent compositions.

Regardless of the folder structure used, the folder concept in itself poses no restrictions, nor does it add any clinical meaning to the record - it simply provides a logical navigational structure to the “lumps” of information committed to the record (remembering that inside compositions, there are other means of providing fine-grained structure in entries).

Note that neither the folder names nor the composition names described and illustrated above are part of the openEHR EHR architecture: all such details are provided by archetypes; hence, EHR structures based on completely different conceptions of division of information, or even different types of medicine are equally possible.

The folder structure of an EHR constitutes a third category of information that must be controlled over time, in order to allow changes to the folder structure to be remembered along with changes to content. The contents of a typical EHR now resemble FIGURE 7.

FIGURE 7 An EHR containing Compositions and Folder Structure CIs

4.2.3 EHR Status

A final category of information which may be needed in the change-controlled EHR is that of “EHR status information”, including status flags on the EHR to indicate whether it is actively in use, inactive, and so on. An EHR Status object is used to record such flags. Other EHR-wide meta-data may be recorded in the archetyed part of this object, including environment settings, software application names and version ids, identification and versions of data resources such as terminologies and possibly even actual software tools, configuration files, keys and so on. Such information is commonly versioned in software configuration management systems, in order to enable the reconstruction of earlier versions of software with the correct tools. One reason to store such information at all is that it adds to medico-legal support when clinicians have to justify a seemingly bad decision: if it can be shown that the version of software in use at the time was faulty, they are protected, but to do this requires that such information be recorded in the first place. We therefore add a fourth category of content to the notional controlled EHR - that of “EHR status”, as shown in FIGURE 8.

FIGURE 8 A Comprehensive Medico-legal EHR
4.2.4 Change Control of the EHR

The EHR described thus far is essentially an EHR status object plus a logical arrangement of Compositions within a directory structure of Folders. However we have not described the semantics of update to the record, or audit-trailing of changes.

A number of requirements and design considerations lead to the final design of Compositions in the openEHR EHR. The first is the system interaction context, in which a healthcare agent (usually a human, but may be a software process) interacts with the EHR system to enter data. This is the context during which Compositions and Folder structures are created or modified on the system, and, the contextual details of who, when and where must be recorded. As described in [5], these details will be different from the context details of the healthcare event, as is the case when the EHR is updated some time after a contact, and by other personnel.

Given an EHR in which there is a folder structure, and event and persistent Compositions, the general model of update of the EHR is that any of these might be created and/or modified during the update. The simplest, most common case is probably the creation of a single contact Composition, which is placed in an “events” folder. Another common case will be the creation of an event Composition, and modification of one or more persistent Compositions, e.g. due to facts learned in the consultation about family history, or due to prescription of new medications. Other types of updates include corrections to existing Compositions, and acquisition of compositions from another site such as a hospital. Any of these updates might also include a change to the folder structure, or the moving of existing Compositions to other Folders. Naturally these scenarios depend on a structure of the record including event and persistent compositions, and a folder structure; in the extreme, an EHR consisting only of event Compositions and no folders will experience only the creation of a single Composition for most updates, with acquisitions being the exception.

Recording of contextual information is not the only requirement of the EHR. Numerous projects (GEHR/Europe, GEHR/Australia, SynEx, Synapses etc) as well as standards (CEN 13606, the emerging ISO 18308 EHRRA Requirements) and academic work all agree on the need to satisfy a number of medico-legal requirements of the EHR. These are as follows: that all additions and changes to the record be audit-trailed and that all previous states of the record be available for the purposes of medico-legal investigation. The former is satisfied by the recording of context details in the relevant places (including at the Entry level, dealt with later in this specification). The latter requirement leads to the use of version control of information items, and eventually, to a formal change management approach.

Change management of information is a non-trivial business, and requires a well-defined approach, such as the “configuration managment” (CM) paradigm described in the openEHR Common Information Model. Under this paradigm we can visualise how changes occur to the EHR. FIGURE 9 shows a number of Contributions to the EHR as follows:

- The first is due to a patient contact, and causes the creation of a new contact composition; it also causes changes to the problem list, current medications and care plan compositions (once again, in a differently designed record, all this information might have been contained in a single event Composition; likewise, it might have been distributed into even more Compositions).
- The next Contribution is the acquisition of test results from a pathology laboratory.
- The third is another contact in which both family history and the folder structure are modified.
- This fourth is an error correction (e.g. a misspelled name, wrongly entered value), and shows that there can be a Contribution even when there is no healthcare event.
4.2.5 Versioning of Compositions

Versioning of Compositions is achieved with the \texttt{VERSIONED\_OBJECT\textless T\textgreater} type from the \texttt{change\_control} package (Common IM), which in the \texttt{composition} package is explicitly bound to the \texttt{COMPOSITION} class, via the class \texttt{VERSIONED\_COMPOSITION} which inherits from the type \texttt{VERSIONED\textless\textgreater COMPOSITION>}. 

- The last is an update to the EHR status information in the EHR, due to a software upgrade.

We can now see that the configuration management paradigm is a suitable approach. Consider the EHR described thus far:

- it is a repository of information about the patient, which is separated into distinct entities, (Compositions);
- it may have a directory of Folders acting as a navigational structure of the compositions;
- there can be multiple, simultaneous users of the repository;
- changes to the information occur due to interactions of multiple users with the repository, and must be attested with revision history information;
- previous states in time of the repository must be available upon request.

Thus, the EHR corresponds very closely to the general model of a change-controlled repository. The implication is that we should consider contributions to the EHR to be the change sets described in the CM paradigm, where each contribution causes the creation or modification of one or more Compositions (configuration items, or “CIs”) and/or changes to the directory structure and status information.
The effect of version control on Compositions is visualised in FIGURE 10. The versions (each “version” being a COMPOSITION) shown here in a VERSIONED_COMPOSITION are the same versions shown along each vertical line in FIGURE 9, this time shown with their associated audit items. The set of versions should be understood as a set of successive modifications of the same data in time.

The VERSIONED_COMPOSITION can be thought of as a kind of smart repository: how it stores successive versions in time is an implementation concern (there are a number of intelligent algorithms available for this sort of thing), but what is important is that its functional interface enables any version to be retrieved, whether it be the latest, the first, or any in between.

Returning to Compositions, the logical types “event Composition” and “persistent Composition” are modelled using the class COMPOSITION, a coded attribute category, and a Boolean-returning function is_persistent. The function returns true for those categories deemed to be persistent categories, and can be implemented as required on each system. For event Compositions this function should return false. There will usually be a context attribute, which carries the clinical context information corresponding to the event, although in continuous care situations this might not always be used. Persistent Compositions do not include this, although if it is necessary to determine what clinical session (if any) caused an update to a persistent Composition, the Contribution can be found from the audit, and then checked for the presence of an event Composition.

4.2.6 Versioning Scenarios

The following scenarios for creating new COMPOSITION versions have been identified as follows.

*Case 0*: information is authored locally, causing the creation of a new VERSION<COMPOSITION>. If this is the first version, a new VERSIONED_COMPOSITION will be created first.

*Case 1*: information is modified locally, such as for the correction of a wrongly entered datum in a composition. This causes the creation of a new VERSION<COMPOSITION> in an existing VERSIONED_COMPOSITION, in which the AUDIT_DETAILS.change_type is set to “correction”.

*Case 2*: information received from a feeder system, e.g. a test result, which will be converted and used to create a new VERSION<COMPOSITION>. This kind of acquisition could be done automatically. If the receiver system needs to store a copy of the original feeder system audit details, it writes it into the COMPOSITION.feeder_audit.

*Case 3*: a VERSION<COMPOSITION> (such as a family history) received as part of an EHR_EXTRACT from another openEHR system, which will be used by a local author to create a new COMPOSITION that includes some content chosen from the received item. In this case, the new VERSION<COMPOSITION> is considered as a locally authored one, in which some content has been obtained from elsewhere. If it is the first version, a
VERSIONED_COMPOSITION is first created. The AUDIT_DETAILS documents the committal of this content, and the clinician may choose to record some details about it in the audit description.

In summary, the AUDIT_DETAILS is always used to document the addition of information locally, regardless of where it has come from. If there is a need to record original audit details, they become part of the content of the versioned object.
## 5 EHR Package

### 5.1 Overview

The ehr package is illustrated in FIGURE 11. The EHR class is the root access point of the health record for a subject of care, and is a change-controlled repository of the kind described in the openEHR Common Information Model. Accordingly, it contains the identifiers of various versioned objects, as well as the list of Contributions made to it. The versioned objects consist of:

- an EHR Status object, in the form of a `VERSIONED_EHR_STATUS` instance;
- a directory, in the form of a `VERSIONED_FOLDER` instance (defined in the Common IM);
- Compositions in terms of `VERSIONED_COMPOSITIONs`.

The root EHR object is very simple, and records only the identifier of the system where it was created, the identifier of the EHR (distinct from any identifier for the subject of care), and the time of creation. Otherwise, it simply acts as an access point for the content items and Contributions of the EHR.

A single EHR Status object is included in every openEHR EHR, and provides a place where EHR-wide settings, flags and environmental information can be recorded. It is versioned, so that any change to any part of the content is medico-legally auditable.

The main data of the EHR consists of its Compositions. References rather than containment by value are used for the `compositions` relationship between the EHR and `VERSIONED_COMPOSITION` classes, reflecting the vast majority of retrieval scenarios in which only select (usually recent) Compositions are needed. Containment by value would lead to systems which retrieved all `VERSIONED_COMPOSITION` objects every time the EHR object was accessed.

The directory structure is optional, and consists of FOLDERs (defined in the Common IM, directory package), enabling the construction of a hierarchical directory of any complexity. Each Folder...
in the structure can contain any number of references to versioned Compositions. The structure as a whole acts as a directory for organising versioned Compositions in the record.

The use of references between Folders and Compositions allows more than one Folder to refer to the same Composition, thus allowing multiple ways of finding or classifying Compositions. This arrangement is akin to a computer file system that allows links (Unix) or “shortcuts” (Windows); in the EHR it allows for example a contact Composition to be grouped with other Compositions in an episode, and also in a group corresponding to a type of problem. The sophistication of the folder structure is completely under the control of the designers of EHR systems, and can be as simple or complex as required, according to the use of archetypes. The whole Folder structure may correspond to one archetype, or there may be multiple archetypes used to create it.

The same logic holds for the relationship between the EHR and its CONTRIBUTIONS. The role of Contributions, as documented in the Common Information Model, is to record the set of versions added to a repository during a single logical update, along with the audit details of the change. In the context of the EHR as one such repository, each instance of the CONTRIBUTION class potentially includes in its list of versions not only compositions, but also the folder structure itself, if this was changed during the update. Since the versions attribute of the CONTRIBUTIONS class is in fact a list of OBJECT_REFS, the same list can accommodate both types of object.

### 5.2 EHR Creation Semantics

When an EHR is created, the result should be a root EHR object and an EHR Status object, plus any other house-keeping information the versioning implementation requires. In a normal implementation, the EHR Status object would normally be created and committed in a Contribution, just as any Composition would be. The EHR Status object has a special status in the EHR, indicating whether the EHR should be included in querying, whether it is modifiable, and by implication, whether it is active. Flags might be set to indicate that it is test record, or for educational or training purposes. The initial creation operation has to supply sufficient parameters for creation of these two objects, including:

- system id
- EHR id
- Subject id (optional; the use of PARTY_SELF allows completely anonymous EHRs)
- queryable flag
- modifiable flag
- any other flags required by the EHR Status object in the local implementation.

### 5.3 Time in the EHR

There are numerous times recorded in the EHR, at varying levels of granularity. The following figure indicates the main ones common to Observation recordings (generally the most numerous) in the
EHR. These times have a close correspondence with the contexts described in Context Model of Recording on page 16.

5.4 Historical Views of the Record

It is important to understand that the COMPOSITION versions at a previous point in time represent a previously available informational state of the EHR, at a particular EHR node. Such previous states include only those Compositions from other sources as have been acquired by that point in time, regardless of whether the acquired information pertains to clinical information recorded earlier. A previous historical state of the EHR thus corresponds to what users of a system could see at a particular moment of time. It is important to differentiate this from previous clinical states of the patient: previous informational states of the EHR might include acquired information which is significantly older than the point in time when merging occurred. A previous clinical state of the patient would be a derivable view of the EHRs in all locations for the patient - what is sometimes called the virtual EHR - at a given point in time, minus acquired Compositions, since these constitute (usually out-of-date) copies of Compositions primarily available elsewhere.

It is previous informational states with which we are concerned for medico-legal purposes, since they represent the information actually available to clinicians at a health-care facility, at a point in time. But previous clinical views may be useful for reconstructing an actual sequence of events as experienced by the patient.

5.5 Class Descriptions

5.5.1 EHR Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The EHR object is the root object and access point of an EHR for a subject of care.</td>
</tr>
<tr>
<td>CEN</td>
<td>EHCR class</td>
</tr>
<tr>
<td>Synapses</td>
<td>RecordFolder class</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_EHR</td>
</tr>
</tbody>
</table>

FIGURE 12 Time in the EHR
### Versioned EHR Status Class

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>system_id</td>
<td>HIER_OBJECT_ID</td>
<td>The id of the EHR system on which this EHR was created.</td>
</tr>
<tr>
<td>ehr_id</td>
<td>HIER_OBJECT_ID</td>
<td>The id of this EHR.</td>
</tr>
<tr>
<td>time_created</td>
<td>DV_DATE_TIME</td>
<td>Time of creation of the EHR</td>
</tr>
<tr>
<td>contributions</td>
<td>List&lt;OBJECT_REF&gt;</td>
<td>List of contributions causing changes to this EHR. Each contribution contains a list of versions, which may include references to any number of VERSION instances, i.e. items of type VERSIONED_COMPOSITION and VERSIONED_FOLDER.</td>
</tr>
<tr>
<td>ehr_status</td>
<td>OBJECT_REF</td>
<td>EHR_STATUS object for this EHR.</td>
</tr>
<tr>
<td>directory</td>
<td>OBJECT_REF</td>
<td>Optional directory structure for this EHR.</td>
</tr>
<tr>
<td>compositions</td>
<td>List&lt;OBJECT_REF&gt;</td>
<td>Master list of all composition references in this EHR</td>
</tr>
</tbody>
</table>

#### Invariants

- **System_id_valid**: system_id /= Void
- **Ehr_id_valid**: ehr_id /= Void
- **Time_created_valid**: time_created /= Void
- **Contributions_valid**: contributions /= Void and then contributions.for_all(type.is_equal("CONTRIBUTION"))
- **Compositions_valid**: compositions /= Void and then compositions.for_all(type.is_equal("VERSIONED_COMPOSITION"))
- **Directory_valid**: directory /= Void implies directory.type.is_equal("VERSIONED_FOLDER")

### 5.5.2 Versioned EHR Status Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSIONED_EHR_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Versioning container for EHR_STATUS instance.</td>
</tr>
<tr>
<td>Inherit</td>
<td>VERSIONED_OBJECT&lt;EHR_STATUS&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariants</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 5.5.3 EHR_STATUS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EHR_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Single object per EHR giving various EHR-wide information.</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject: PARTY_SELF</td>
<td>The subject of this EHR. The external_ref attribute can be used to contain a direct reference to the subject in a demographic or identity service. Alternatively, the association between patients and their records may be done elsewhere for security reasons.</td>
</tr>
<tr>
<td>is_queryable: Boolean</td>
<td>True if this EHR should be included in population queries, i.e. if this EHR is considered active in the population.</td>
</tr>
<tr>
<td>is_modifiable: Boolean</td>
<td>True if this EHR is allowed to be written to.</td>
</tr>
<tr>
<td>other_details: ITEM_STRUCTURE</td>
<td>Any other details of the EHR summary object, in the form of an archetyped Item_structure.</td>
</tr>
</tbody>
</table>

#### Invariants

- Is_archetype_root: is_archetype_root
- Subject_valid: subject /= Void
- No_parent: parent = Void

### 5.5.4 VERSIONED_COMPOSITION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSIONED_COMPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Version-controlled composition abstraction, defined by inheriting VERSIONED_OBJECT&lt;COMPOSITION&gt;.</td>
</tr>
<tr>
<td>GEHR</td>
<td>GI_VERSIONED_COMPOSITION</td>
</tr>
<tr>
<td>Inherit</td>
<td>VERSIONED_OBJECT&lt;COMPOSITION&gt;</td>
</tr>
</tbody>
</table>

#### Function

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_persistent: Boolean</td>
<td>Indicates whether this composition set is persistent; derived from first version.</td>
</tr>
</tbody>
</table>
### Invariants

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSIONED_COMPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Archetype_node_id_valid</strong></td>
<td><code>all_versions.for_all((c:COMPOSITION) c.archetype_node_id.is_equal(all_versions.first.archetype_node_id))</code></td>
</tr>
<tr>
<td><strong>Persistent_valid</strong></td>
<td><code>all_versions.for_all((c:COMPOSITION) c.is_persistent = all_versions.first.data.is_persistent)</code></td>
</tr>
<tr>
<td><strong>Owner_id_valid</strong></td>
<td><code>owner_id.generating_type.is_equal(&quot;EHR&quot;)</code></td>
</tr>
</tbody>
</table>
6 Composition Package

6.1 Overview

The Composition is the primary ‘data container’ in the openEHR EHR and is the root point of clinical content. Instances of the Composition class can be considered as self-standing data aggregations, or documents in a document-oriented system (similar to HL7 CDA Documents). The key information in a Composition is found in its content, context, and composer attributes. The majority of the use of paths in openEHR is likely to be within Compositions. FIGURE 13 illustrates the composition package.

6.2 Design Principles

6.2.1 Composition Composer

The composer is the person who was primarily responsible for the content of the Composition. This is the identifier which should appear on the screen. It could be a junior doctor who did all the work, even if not legally responsible, or it could be a nurse, even if later attested by a more senior clinician; it will be the patient in the case of patient-entered data. It may or may not be the person who entered and committed the data. It may also be a software agent. This attribute is mandatory, since all content must be been created by some person or agent.

Since in many cases Compositions will be composed and committed by the same person, it might seem that two identifiers COMPOSITION.composer and VERSION.audit.committer (which are both of type PARTY_PROXY) will be identical. In fact, this will probably not be the case, because the kind of identifier to represent the composer will be a demographic identifier, e.g. “RN Jane Williams”, “RN
12345678”, whereas the identifier in the audit details will usually be a computer system user identifier, e.g. “jane.williams@westmead.health.au”. This difference highlights the different purposes of these attributes: the first exists to identify the clinical agent who created the information, while the second exists to identify the logged-in user who committed it to the system.

In the situation of patient-entered data, the special “self” PARTY_PROXY instance (see Common IM generic package) is used for both COMPOSITION.composer and VERSION.audit.committer.

6.2.2 Event Context

Overview
The optional event_context in the COMPOSITION class is used to document the healthcare event causing new or changed content in the record. Here, ‘healthcare event’ means ‘a (generally billable) business activity of the healthcare system with, for or on behalf of the patient’. Generally this will an encounter involving the subject of care and physician, but is variable in a hospital environment. In this sense, a visit to a GP is a single care event, but so is an episode in a hospital, which may encompass multiple encounters. The information recorded in Event context includes start and (optional) end time of the event, health care facility, setting (e.g. primary care, aged care, hospital), participating healthcare professionals, and optional further details defined by an archetype.

Healthcare events that require an Event_context instance in their recorded information include the following.

- Scheduled or booked patient encounters leading to changes to the EHR, including with a GP, hospital consultant, or other clinical professional such as mobile nurse. In this case, the Event context documents the time and place of the encounter, and the identity of the clinical professional.
- Case conferences about a patient, leading to modifications to the health record; here the Event context documents the case conference time, place and participants.
- Pathology, imaging or other test process. In this case, the Event context documents the place and period during which testing and analysis was carried out, and by whom.
- Data resulting from care in the home provided by health professional(s) (often allied health care workers).

Situations in which Event context is optional include the following.

- Nurse interactions with patients in hospital, including checking vital signs, adjusting medication or other aspects of bed situation for the patient. Each instance of a nurse’s observations are generally not considered to be a separate ‘care event’, rather they are seen as the continuation of the general activities of monitoring. In such situations, the overall context is given by ADMIN_ENTRY instances in the record indicating date/time and place of admission and discharge.

Situations in which Event context is not used include:

- Any modification to the EHR which corrects or updates existing content, including by administrative staff, and by clinical professionals adding or changing evaluations, summaries etc.
- Patient-entered data where no interaction with health professionals took place; typically readings from devices in the home such as weighing scales, blood glucose measuring devices, wearable monitors etc.

Ultimately, the use of Event context will be controlled by Composition-level archetypes.
Occurrence in Data
For situations requiring an `EVENT_CONTEXT` object to be recorded, it is worth clarifying which Compositions carry such objects. Consider the example shown in FIGURE 14. In this example, a Contribution is made to the EHR, consisting of one or more Compositions that were each created or modified due to some clinical activity. Within such a set, there will usually be one Composition relating directly to the event, such as the patient contact - this is the Composition containing the doctor’s observations, nurses’ activities etc, during the visit, and is therefore the one which contains the `EVENT_CONTEXT` instance. Other Compositions changed during the same event (e.g. updates to medication list, family history and so on) do not require an Event context, since they are part of the same Contribution, and the event context of the primary Composition can always be retrieved if desired. Contributions A, B and C in the figure illustrate this case.

In cases where Contributions are made to the record with no event context, the Event context of any Compositions from the original commit will remain intact and visible (unless the correction is to the event context itself of course), and will correctly reflect the fact that no new clinical interaction occurred. This is the case with Contribution D in the figure.

Persistent Compositions do not have an Event context.

Time
The times recorded in the Event context represent the time of an encounter or other activity undertaken by a health provider to/for/on behalf of the patient. The time is represented as a mandatory start time and optional end time. It is assumed that where there is a clinical session (i.e. an `EVENT_CONTEXT` object does exist), the start time is known or can be reasonably approximated. It is...
quite common that the end time of a consultation or encounter is not recorded, but rather inferred from e.g. average consult times, or the start time of the next consult for the same physician.

Event context is used as described above even if the additions are made to the EHR long after the event took place, such as happens when a doctor writes his/her notes into the record system at night, after all patients have been seen. In such cases, the versioned Composition audit trail records the context of when the data were entered, as distinct from the context of when the clinical interaction took place.

**Participations**

As part of the Event context, the participations attribute can be used to describe who participated, and how. Each participation object describes the “mode” of participation as well, such as direct presence, video-conference and so on. In many cases such information is of no interest, since the subject of any Entry is known (ENTRY.subject) and the clinician will be known (COMPOSITION.composer), and the mode of communication is assumed to be a personal encounter. The participations attribute is therefore used when it is desired to record further details of how the patient and or physician interacted (e.g. over the internet), and/or other participants, such as family, nurses, specialists etc.

There are no general rules about who is included as a participant. For example, while there will be a patient participation during a GP visit, there will be no such participation recorded when the clinical event is a tissue test in a laboratory. Conversely, a patient might record some observations and drug self-administration in the record, in which case the composer will be the patient, and there will be no clinician participation. Consequently, the use of participations will mostly be archetype-driven.

**Healthcare Facility, Location and Setting**

The health_care_facility (HCF) attribute is used to record the health care facility in whose care the event took place. This is the most specific identifiable (i.e. having a provider id within the health system) workgroup or care delivery unit within a care delivery enterprise which was involved in the care event. The identification of the HCF can be used to ensure medico-legal accountability. Often, the HCF is also where the encounter physically took place, but not in the case of patient home visits, internet contacts or emergency care; the HCF should not be thought of as a physical place, but as a care delivery management unit. The physical place of care can be separately recorded in EVENT_CONTEXT.location. The health_care_facility attribute is optional to allow for cases where the clinical event did not involve any care delivery enterprise, e.g. self-care at home by the patient, emergency revival by a non-professional (e.g. CPR by lifeguard on a beach), care by a professional acting in an unofficial capacity (doctor on a plane asked to aid a passenger in difficulty). In all other cases, it is mandatory. Archetypes are used to control this.

Two other context attributes complete the predefined notion of event context in the model: location and setting. The location attribute records: the physical location where the care delivery took place, and should document a reasonably specific identifiable location. Examples include “bed 5, ward E”, “home”. This attribute is optional, since the location is not always known, particularly in legacy data.

The setting attribute is used to document the “setting” of the care event. In clinical record keeping, this has been found to be a useful coarse-grained classifier of information. The openEHR Terminology “setting” group is used to code this attribute. It is mandatory, on the basis that making it optional will reduce its utility for querying and classification.

### 6.2.3 Composition Content

The data in a Composition is stored in the content attribute. There are four kinds of data structuring possible in the content attribute:
it may be empty. Although for most situations, there should be content in a Composition, there are at least two cases where an empty Composition makes sense:
- the first is a Composition in ‘draft’ editing state (VERSION.lifecycle_state = ‘incomplete’)
- the second is for systems that are only interested in the fact of an event having taken place, but want no details, such as so-called clinical ‘event summary’ systems, which might record the fact of visits to the doctor, but contain no further information. This can be achieved using Compositions with event context, and no further content.

it may contain one or more SECTIONs which are defined in the archetype of the Composition;
it may contain one or more SECTION trees, each of which is a separately archetyped structure;
it may contain one of more ENTRYs directly, with no intermediate SECTIONs;
it may be any combination of the last three possibilities.

The actual structures used in a Composition at runtime are controlled by a template, which in turn controls the particular combination of archetypes used.

### 6.3 Path Semantics

### 6.4 Class Descriptions

#### 6.4.1 COMPOSITION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>COMPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>One version in a VERSIONED_COMPOSITION. A composition is considered the unit of modification of the record, the unit of transmission in record extracts, and the unit of attestation by authorising clinicians. In this latter sense, it may be considered equivalent to a signed document.</td>
</tr>
<tr>
<td>CEN</td>
<td>Composition</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_COMPOSITION_VERSION</td>
</tr>
<tr>
<td>Synapses</td>
<td>Composition class</td>
</tr>
<tr>
<td>HL7v3</td>
<td>CDA DOCUMENT</td>
</tr>
</tbody>
</table>

Inherit LOCATABLE

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>content: List&lt;CONTENT_ITEM&gt;</td>
<td>The content of this Composition.</td>
</tr>
</tbody>
</table>
### 6.4.2 EVENT_CONTEXT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>COMPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>context: EVENT_CONTEXT</td>
</tr>
<tr>
<td>1..1</td>
<td>composer: PARTY_PROXY</td>
</tr>
<tr>
<td>1..1</td>
<td>category: DV_CODED_TEXT</td>
</tr>
<tr>
<td>1..1</td>
<td>territory: CODE_PHRASE</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 is_persistent: Boolean</td>
<td>True if category is a “persistent” type, False otherwise. Useful for finding Compositions in an EHR which are guaranteed to be of interest to most users.</td>
</tr>
</tbody>
</table>

### Invariants

- **Is_archetype_root**: is_archetype_root
- **Composer_valid**: composer /= Void
- **Content_valid**: content /= Void implies not content.is_empty
- **Category_validity**: category /= Void and then terminology(“openehr”).codes_for_group_name(“composition category”, “en”).has(category.defining_code)
- **Is_persistent_validity**: is_persistent implies context = Void
- **Territory_valid**: territory /= Void and then code_set(“countries”).has(territory)
- **No_parent**: parent = Void
### Purpose
Documents the context information of a healthcare event involving the subject of care and the health system. The context information recorded here are independent of the attributes recorded in the version audit, which document the “system interaction” context, i.e. the context of a user interacting with the health record system. Healthcare events include patient contacts, and any other business activity, such as pathology investigations which take place on behalf of the patient.

### CEN
Composition class

### Synapses
Composition class

### HL7v3
TBD

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>health_care_facility:</td>
<td>The health care facility under whose care the event took place. This is the most specific workgroup or delivery unit within a care delivery enterprise that has an official identifier in the health system, and can be used to ensure medico-legal accountability.</td>
</tr>
<tr>
<td></td>
<td>PARTY_IDENTIFIED</td>
<td></td>
</tr>
<tr>
<td>1..1</td>
<td>start_time: DV_DATE_TIME</td>
<td>Start time of the clinical session or other kind of event during which a provider performs a service of any kind for the patient.</td>
</tr>
<tr>
<td>0..1</td>
<td>end_time: DV_DATE_TIME</td>
<td>Optional end time of the clinical session.</td>
</tr>
<tr>
<td>0..1</td>
<td>participations:</td>
<td>Parties involved in the healthcare event. These would normally include the physician(s) and often the patient (but not the latter if the clinical session is a pathology test for example).</td>
</tr>
<tr>
<td></td>
<td>List &lt;PARTICIPATION&gt;</td>
<td></td>
</tr>
<tr>
<td>0..1</td>
<td>location: String</td>
<td>The actual location where the session occurred, e.g. “microbiol lab 2”, “home”, “ward A3” and so on.</td>
</tr>
<tr>
<td>1..1</td>
<td>setting: DV_CODED_TEXT</td>
<td>The setting in which the clinical session took place. Coded using the openEHR Terminology, “setting” group.</td>
</tr>
<tr>
<td>0..1</td>
<td>other_context: ITEM_STRUCTURE</td>
<td>Other optional context which will be archetyped.</td>
</tr>
</tbody>
</table>
### Invariants

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EVENT_CONTEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>start_time_valid</strong>:</td>
<td>start_time /= Void</td>
</tr>
<tr>
<td><strong>participations_validity</strong>:</td>
<td>participations /= Void implies not participations.is_empty</td>
</tr>
<tr>
<td><strong>location_valid</strong>:</td>
<td>location /= Void implies not location.is_empty</td>
</tr>
<tr>
<td><strong>setting_valid</strong>:</td>
<td>setting /= Void and then Terminology(“openehr”).codes_for_group_name(“setting”, “en”).has(setting.defining_code)</td>
</tr>
</tbody>
</table>
7 Content Package

7.1 Overview

The content package contains the CONTENT_ITEM class, ancestor class of all content types, and the navigation and entry packages, which contain SECTION, ENTRY and related types. The content package is illustrated in FIGURE 15.

![FIGURE 15 rm.composition.content package]

7.2 Class Descriptions

7.2.1 CONTENT_ITEM Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CONTENT_ITEM (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract ancestor of all concrete content types.</td>
</tr>
<tr>
<td>Inherit</td>
<td>CONTENT_ITEM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>
8 Navigation Package

8.1 Overview

The navigation Package defines a hierarchical heading structure, in which all individual headings are considered to belong to a “tree of headings”. Each heading is an instance of the class SECTION, illustrated in FIGURE 16.

Sections provide both a logical structure for the author to arrange Entries, and a navigational structure for readers of the record, whether they be human or machine. Sections are archetyped in trees with each tree containing a root Section, one or more sub-sections, and any number of Entries at each node. Section trees that are separately archetyped, such as the SOAP headings, or the heading structure for a physical examination, can be combined at runtime by type to form one large heading structure, as shown in FIGURE 17.

In terms of understanding of clinical data, Section structures are not essential in a Composition - they can always be removed or ignored (typically in machine processing such as querying) without losing the meaning of the Entries in the Composition. While Sections are often used to group Entries according to status, e.g. “family history”, “problems”, “observations”, it is the Entries themselves that indicate the definitive category of information contained therein. This principle is explained in more detail in Entry and its Subtypes on page 50.

Despite the above, Section structures do not have to be regarded as ad hoc or unreliable structures. On the contrary, as they are archetyped, their structures can be relied upon in the same way as any other structure in the record can be relied on to conform to its archetype. Accordingly, solid assumptions can be made about Sections, based on their archetypes, for the purposes of querying. In fact, the main benefit of Sections is that they may provide significant performance benefits to querying, whether by interactive application or by automated systems.

One potentially confusing aspect of any Section structure is that while the root Section in a tree is logically a Section, it would not appear in a display or printed form as a visible section, due to the fact that humans don’t usually write down top-level headings for anything, since there is always a containing structure acting as a top-level organising context (such as the piece of paper one is writing on). For example, consider the way a clinician writes down the problem/SOAP headings on paper. She writes the name of the first problem, then under that, the S/O/A/P headings, then repeats the process for further problems. But she doesn’t write down a heading above the level of the problems, even though there must be one from a data structure point of view.
8.2 Class Descriptions

8.2.1 SECTION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents a heading in a heading structure, or “section tree”.</td>
</tr>
<tr>
<td>Use</td>
<td>Created according to archetyped structures for typical headings such as SOAP, physical examination, but also pathology result heading structures.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Should not be used instead of ENTRY hierarchical structures.</td>
</tr>
<tr>
<td>CEN</td>
<td>Headed_section</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::CompositeObservation (COAS does not distinguish between the semantics of Sections and hierarchical structure inside Entries, modelled in openEHR by Cluster).</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_ORGANISER</td>
</tr>
<tr>
<td>HL7v3</td>
<td>CDA Heading.</td>
</tr>
<tr>
<td>Inherit</td>
<td>CONTENT_ITEM</td>
</tr>
</tbody>
</table>

FIGURE 17 Section View of a General Practice Contact Composition
### 8.3 Section Instance Structures

#### 8.3.1 Problem/SOAP Headings

An example of an section tree representing the problem/SOAP heading structure is shown in FIGURE 18.

<table>
<thead>
<tr>
<th>CLASS</th>
<th>SECTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
</tbody>
</table>
| 0..1 | items: List<CONTENT_ITEM> | Ordered list of content items under this section, which may include:  
• more SECTIONs  
• ENTRYs |

| Invariants | Items_valid: items /= Void implies not items.is_empty |

---

**Items_valid**: items /= Void implies not items.is_empty
FIGURE 18 “problem/SOAP” Section Structure
9 Entry Package

9.1 Design Principles

9.1.1 Information Typology

All information which is created in the openEHR health record is expressed as an instance of a class in the entry package, containing the ENTRY class and a number of descendants. An ENTRY instance is logically a single ‘clinical statement’, and may be a single short narrative phrase, but may also contain a significant amount of data, e.g. an entire microbiology result, a psychiatric examination, a complex prescription. In terms of actual content, the Entry classes are the most important in the openEHR EHR Information Model, since they define the semantics of all the ‘hard’ information in the record. They are intended to be achetyped, and in fact, archetypes for Entries make up the vast majority of important clinical archetypes defined.

The design of Entry package is based on past experience and upon a solid theoretical model of clinical recording, described in Beale & Heard [4], which includes a typology of information categories shown in the FIGURE 19 (the openEHR Entry class names are annotated next to categories which are directly modelled).

The key top-level categories in the typology are ‘care information’ and ‘administrative information’. The former encompasses all statements that might be recorded at any point during the care process, and consists of the major subcategories ‘observation’, ‘opinion’ and ‘instruction’, which themselves correspond to the past, present and future in time (ISO TC215 uses the terms ‘retrospective’, ‘current’ and ‘prospective’). The administrative information category covers information which is not generated by the care process proper, but relates to organising it, such as appointments and admissions. This information is not about care, but about the logistics of care being delivered. Categories that relate to the patient system as observed and understood are shown as white bubbles, while categories that relate to intervention into the patient system are shown as shaded. The opinion category has features of both passive analysis and active intervention.

FIGURE 19 Typology of Recorded Information (adapted from Beale & Heard, 2006)

The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
Where the principal information categories of FIGURE 19 occur in the care delivery process is shown in FIGURE 20. This figure shows the cycle of information in an iterative, investigation process typical not just of clinical medicine, but of science itself. The system as a whole is made up of two parts: the “patient system” and the “clinical investigator system”. The latter consists of health carers, and may include the patient, and is responsible for understanding the state of the patient system and delivering care to it.

Starting from the patient (right hand side of figure) we can see that Observations lead to Opinions on the part of the investigator, including assessment of the current situation, goals for a future situation, and plans for achieving the goals. The latter lead to Instructions designed to help the patient achieve the goals. A complex or chronic problem may take numerous iterations - possibly a whole lifetime’s worth - with each step being quite small, and future steps depending heavily on past progress. The role of the investigator (and associated agents) is normally filled by health care professionals, but may also be filled by the patient, or a guardian or associate of the patient. Indeed, this is what happens every time a person goes home from the pharmacy with prescribed medication to take at home.

There are two key justifications for using such a typology as a basis for class design. Firstly, although for all categories in the typology there is a meaning for ‘contextual’ attributes of time, place, identity, reason and so on, each category has a different structure for these attributes. For example, time in the Observation category has a linear historical structure, whereas in Instruction it has a branching, potentially cyclic structure. The separation of types allows these contextual attributes to be modelled according to the type. Secondly, the separation of types provides a systematic solution to the so-called problem of “status” or “meaning modification” of clinical statement values, as described below.

**9.1.2 Clinical Statement Status**

A well-known problem in clinical information recording is the problem of assigning “status”, including variants like “actual value of P” (P stands for some phenomenon), “family history of P”, “risk of P”, “fear of P”, as well as negation of any of these, i.e. “not/no P”, “no history of P” etc. A proper analysis of these so called statuses [4] shows that they are not “statuses” at all, but different categories of information as per the typology. The common statement types mentioned here are mapped as follows:

- actual value of P ⇒ Observation (of P);
- no/not P ⇒ Observation (of excluded P or types of P, e.g. allergies).
- family history of P ⇒ Evaluation (that patient is at risk of P);
- no family history of P ⇒ Evaluation (that P is an excluded risk);
Entry Package
The openEHR EHR Information Model
Rev 5.0

- risk of P \(\Rightarrow\) Evaluation (that patient is at risk of P);
- no risk of P \(\Rightarrow\) Evaluation (that patient is not at risk of P);
- fear of P \(\Rightarrow\) Observation (of FEAR, with P mentioned in the description);

Another set of statement types that can be confused in systems that do not properly model information categories concern interventions, e.g. “hip replacement (5 years ago)”, “hip replacement (planned)”, “hip replacement (ordered for next tuesday 10 am)”. Ambiguity is removed here as well, with the use of the correct information categories, e.g. (I stands for an intervention):

- I (distant past) \(\Rightarrow\) Observation (of I present in patient);
- I (recent past) \(\Rightarrow\) Action (of I having been done to/for patient);
- I (proposed) \(\Rightarrow\) Evaluation, subtype Proposal (of I suggested/likely for patient);
- I (ordered) \(\Rightarrow\) Instruction (of I for patient for some date in the future).

Correct use of the categories from the typology is facilitated by using archetypes designed to map particular kinds of clinical statement to particular ENTRY subtypes. In a system where Entries are thus modelled, there will be no danger of incorrectly identifying the various kinds of Entries, as long as the Entry subtype, time, and certainty/negation are taken into account.

It is worth recalling the high-level division between EHR and demographic information in openEHR. The general approach of openEHR is to enable the complete separation of demographic (particularly patient-identification information) from health records, both in the interests of privacy (in some cases required by national legislation) and separate data management. The Demographic IM therefore defines demographic information. However, there is nothing to prevent certain demographic information occurring in the EHR, and in some cases this is desirable. The two main cases for this are:

- clinically-relevant patient information, such as age, sex, height, weight, eye-colour, ethnicity or ‘race’, occupation;
- identifiers and/or names of health care provider individuals and organisations may be stored directly in the EHR, regardless of whether there is also more detailed information about such entities in the demographic system.

The model of all information intended for a separate openEHR demographic service (itself usually a front-end to an existing hospital master patient index or similar) is defined in the openEHR Demographic Information Model.

9.2 Entry and its Subtypes

The Entry model is defined by the composition.content.entry package, shown in FIGURE 21. The choice of subtypes of ENTRY is based on the typology shown in FIGURE 19 and its associated model. The names do not coincide exactly however, for a number of reasons. Firstly, the category names in the hierarchy were chosen based on a philosophical/scientific model of investigation, and reflecting linguistic norms for the meanings of the terms, whereas the class names used here reflect common health computing and clinical usage of these terms (i.e. names which will make sense to software developers in the health arena). Names in these two cultures do not always coincide. Additionally, for categories such as Opinion and Instruction, the subcategories shown in the typology (e.g. Assessment, Goal and Plan) are too variable to safely be subtyped in software, and are distinguished only at the archetype level. Only a single class for each of these ontological groups is used in the formal model. The use of different names and a slightly simplified mapping has however prevented the faithful implementation of the semantics of the model. The model classes are described in the following subsections.
9.2.1 The Entry class

All Entries have a number of attributes in common. The language and encoding attributes indicate how all text data within the Entry are to be interpreted linguistically and at the character set level. Normally, the language will be the same throughout the entire Entry (if not Composition), but in cases where it is not, the optional language attribute of DV_TEXT can be used to override the value in the enclosing ENTRY (or other enclosing structure, if a DV_TEXT is being used in some other context). Character encoding can be overridden in the same fashion by the encoding attribute within DV_TEXT.

The other attributes common to all Entry subtypes are as follows:

subject: this attribute records the subject of the Entry as an instance of a subtype of PARTY_PROXY. When this is the record subject, (i.e. the patient), the value is an instance of PARTY_SELF. Otherwise it is typically a family member, sexual partner, or other acquaintance of the record subject. It could also be an organ donor. The latter is expressed in the form of a PARTY_IDENTIFIED or RELATED_PARTY instance, which describes the kind of relationship, and optionally, identifies the demographic entity.

subject_is_self: convenience function returning True when the Entry is about the subject of the record.

provider: the agent who provided the information. This is usually the patient or the clinician, but may be someone else, or a software application or device. If participation details of the provider (e.g. mode of communication) need to be recorded, the details should be recorded once in the EVENT_CONTEXT.participations. The provider attribute is optional, since it is often implicit in the information that was recorded.

other_participations: other participations which existed for this Entry, e.g. a nurse who administered a drug in an INSTRUCTION; only required in cases where participants other than the subject of the information and the provider of the information need to be recorded.

Note that the term ‘provider’ as used here should not be confused with the more specific healthcare term used in many english-speaking countries meaning ‘health care provider’, which is usually understood to be a physician or healthcare delivery enterprise such as a hospital. In the model here, it simply means ‘provider of information’ in the context of an Entry. The information provider is optional, and in many cases will not be recorded, since it will be obvious from the composer and other_participations of the enclosing Composition. In many cases, it is not sensible to record a provider, e.g. in the most mundane case where a GP asks “where does it hurt” and the patient says “here” - in such cases, it can only be considered mutual. It is expected to be used only when the composer of the Composition really needs to specify the origin of specific statements, such as in the following circumstances:

- the information provider is specifically accountable for the Entry data (it is their opinion, their decision, they carried out the test etc) - they might also need to attest it;
- the information provider is an authoritative source, or has provided information from a unique perspective (e.g. the view of a spouse/carer on the patient's functional health status or mental state);
- the information provider’s view might not reflect the consensus (e.g. a patient opinion not held by the composer, a difference between father and mother on a description of a child's sleeping pattern);
- the information provider is not one of the Composition-level participants (e.g. an outside information provider such as someone telephoned during the encounter to provide a lab result, or an automated measurement device, or a decision support software component).
9.2.2 Care_entry and Admin_entry

A basic division occurs between clinical and non-clinical information. The CARE_ENTRY class is an abstract precursor of classes that express information of any clinical activity in the care process around the patient, while ADMIN_ENTRY is used to capture administrative information. The division may occasionally seem ambiguous at a theoretical level, but at a practical level, it is almost always clear. Administrative information has the following characteristics:

- it is created by non-clinical staff, or clinical staff acting in an administrative capacity (e.g. a nurse or doctor who has to fill out an admission form in A&E);
- it expresses details to do with coordinating the clinical process, by recording e.g. admission information (enables clinical staff to know who is in the hospital and where they can be found), appointments (ensuring patient and physician get together at an agreed time and place), discharge/dismissal (allowing clinical staff to know that a patient has been sent home healthy, or transferred to another institution), billing and insurance information (where such information is required in the EHR; it may well be in its own system);
- removing administrative information from the EHR would not compromise its clinical integrity, it would simply mean that carers and patients would no longer know when and where they were supposed to meet, or when the patient has entered or left a health care facility.

Conversely, every instance of a CARE_ENTRY subtype is clinically significant, even if it also carries information which might be of interest to other health management functions billing (e.g. ICD10 coded diagnoses), practice management (e.g. date, time and place in an order for day surgery). The CARE_ENTRY type includes two attributes particular to all clinical entries, namely protocol and guideline_id.

These attributes allow the “how” and “why” aspects of any clinical recording to be expressed. Protocol is often recorded for Observations (e.g. staining method in microscopy) and Instructions (e.g.

9.2.3 Observation

Instances of the OBSERVATION class record the observation of any phenomenon or state of interest to do with the patient, including pathology results, blood pressure readings, the family history and social circumstances as told by the patient to the doctor, patient answers to physician questions during a physical examination, and responses to a psychological assessment questionnaire. Observations are distinguished from Actions in that Actions are interventions whereas Observations record only information relating to the situation of the patient, not what is done to him/her.

The significant information of an Observation is expressed in terms of “data”, “state” and “protocol”. The first of these is recorded in the data attribute, defined as follows:

\[\text{data}: \text{the actual datum being recorded; expressed in the form of a History of Events, each of which can be a complex data structure such as a List, Table, Single (value), or Tree, in its own right. Examples include blood pressure, heartrate, ECG traces.}\]

State information can be recorded in the state attribute of Observation, or within the state attribute of each Event in the Observation data attribute (see below and also Data Structures IM for more detailed explanations). The state attribute in Observation is defined as follows:

\[\text{state}: \text{any particular information about the state of the subject of the Entry necessary to correctly interpret the data, which is not already known in the health record (i.e. facts such as the patient being female, pregnant, or currently undergoing chemotherapy). For example, exersion level (resting, post-marathon...), position (lying, standing), post-glucose challenge,}\]
and so on. The form of the state attribute is the same as that of the data attribute: a History of Events of Item_structures.

The inherited protocol attribute is defined as follows:

protocol: details of how the observation was carried out, which might include a particular clinical protocol (e.g. Bruce protocol for treadmill exercise ECG) and/or information about instruments other other observational methods. This information can always be safely omitted from the user interface, i.e. has no bearing on the interpretation of the data.

The detailed semantics of Observation data are described in the following subsections.

9.2.3.1 Timing in Observations

Many health information models express observation time as one or more attributes with names like ‘observation_time’, ‘activity_time’ and so on. The openEHR model departs from this by modelling historical time inside a History/Event structure defined in the data_structures.history package. In short, this package defines the HISTORY class with an origin attribute, and a series of EVENT instances each containing a time attribute. Instantaneous and interval events are distinguished via the EVENT subtypes POINT_EVENT and INTERVAL_EVENT; interval events have the width attribute is set to the duration of the interval.

9.2.3.2 Valid Contents of a History

One of the aims of the model of Observation described here is to represent in the same way single sample and multi-sample time-based data for which measurement protocol is invariant. It is not intended for measurements in “coarse” time taken by different people, different instruments, or with any other difference in data-gathering technique. In these cases, separate, usually single-sample histories are used, usually occurring in distinct container objects (e.g. distinct Compositions, in the EHR).

Accordingly, in the general practice setting, the use of HISTORY will correspond to measurement series which occur during the clinical session (i.e. during a patient contact). In a hospital setting, nurses’ observations might occur in 4-hourly intervals, and there is no well-defined clinical session - simply a series of ENTRYS during the time of the episode. Two approaches are possible here.

- If each Observation is to be committed to the EHR as soon as it is made, the result will be distinct COMPOSITIONs in time, each with its event_context corresponding to the period of the nurse’s presence. Each Composition will contain one or more Observations, each containing in their data a History of one sample of the measured vital sign.
- If Observations are not committed to the EHR immediately, but are stored elsewhere and only committed (say) at the end of each day, then the result will be a single Composition whose event_context corresponds to the total data gathering period, and which contains Observations whose data are multi-event Histories representing the multiple measurements made over the day.

Whether time-based data remain outside the record until a series of desired length is gathered, or entered as it occurs is up to the design of applications and systems; the approach taken should be based on the desired availability of the data in the system in question. If for example, it must be visible in the EHR as soon as the appropriate Compositions are written, then it should be represented as Histories in each relevant Composition; if it need only be available at some much later point in time (e.g. because it is known that no-one but the treating clinician is interested in it), then it can be stored in another system until sufficient items have been gathered for committal to the EHR.

9.2.3.3 Clinical Semantics of Event Time

In most cases, the times recorded in a History (\texttt{HISTORY.origin} and \texttt{EVENT.time, width}) can be thought of as “the times when the observed phenomena were true”. For example, if a pulse of 88bpm is recorded for 12/feb/2005 12:44:00, this is the time at which the heartrate (for which pulse is a surrogate) existed. In such cases, the \texttt{sample} time, and the measuring time are one and the same.

However in cases where the time of sampling is different from that of measurement, the semantics are more subtle. There are two cases. The first is where a sample is taken (e.g. a tissue sample in a needle biopsy), and is tested later on, but from the point of view of the test, the time delay makes no difference. This might be because the sample was immediately preserved (e.g. freezing, placed in a sterile anaerobic transport container), or because even if it decays in some way, it makes no difference to the test (e.g. bacteria may die, but this makes no difference to a PCT analysis, as long as the biological matter is not physically destroyed).

The second situation is when the sample does decay in some way, and the delay is relevant. Most such cases are in pathology tests, where presence of live biological organisms (e.g. anaerobic bacteria) is being measured. The sample time (or ‘collection’ time) must be recorded. Depending on when the test is done, the results may be interpreted differently.

The key question is: what is the meaning of the \texttt{HISTORY.origin} and \texttt{EVENT.time} attributes in these situations? It is tempting to say that their values are (as in other cases) just the times of the actual act of observation, e.g. microscopy, chromatography etc. However, there are two problems with this. Firstly, and most importantly, all physical samples must be understood as being \textit{indirect surrogates for some aspect of the patient state at the time of sampling}, which cannot be observed by direct, instantaneous means in the way a pulse can be taken. This means that no matter when the laboratory work is done, the time to which the result applies is the \texttt{sample} time. It is up to the laboratory to take into account time delays and effects of decay of samples in order to provide a test result which correctly indicates the state of the patient at the time of sampling. The common sense of this is clear when one considers the extreme case where the patient is in a coma or dead (possibly for reasons completely unrelated to the problem being tested for) by the time laboratory testing actually occurs; however, the test result indicates the situation at the point in time when the sample was taken, i.e. when the patient was alive. The second reason is that some kinds of testing are themselves lengthy. For example fungal specimens require 4-6 weeks to confirm a negative result; checks will be made on a daily or weekly basis to find positive growth. However, the result data reported by the laboratory (and therefore the structure of the Observation) is not related to the timing of the laboratory testing; it is reported as being the result for the time of collection of the specimen from the patient.

The meaning therefore of the \texttt{HISTORY.origin} and \texttt{EVENT.time} attributes in openEHR is always the time of sampling. Where delays between sample and measurement times exist and are significant, they are noted in the protocol section of the Observation; such times are modelled in the appropriate archetypes, and taken into account in results.

9.2.3.4 Two ways of Recording State

State information is optional, and is not needed if the data are meaningful on their own. If it is recorded, it can either be as a History of its own (i.e. using the \texttt{OBSERVATION.state} attribute described above), or else as \textit{state} values within the \texttt{EVENT} instances in the \texttt{OBSERVATION.data} History. Both methods are useful in different circumstances. A separate state history is more likely to be used in a correlation study such as a sports medicine study on heartrate with respect to specific types of exercise. In this method, the state information is a History of Events whose times and widths need not match those of the History in the \texttt{data} attribute. The state data under this approach generally
express the condition of the subject in *absolute* terms, i.e. they are standalone statements about the subject’s state at certain points in time, such as “walking on treadmill 10km/h, 10° incline”.

The other method will be used in most general medicine, e.g. for recording fasting and post glucose challenge states of a patient undergoing a glucose tolerance test. (See the Data Structures Information Model for more details). State values stored within the *data* History represent the situation in the subject at the time of the Event within the History and usually in relation to it, for example “post 8 hour fast”. Recording the latter example in an independent state History would require an Event of 8 hours’ duration called ‘fast’. The latter would be technically still correct, but would be very unnatural to most clinicians. FIGURE 22 illustrates the two methods of recording state.

![FIGURE 22 Alternative ways of recording state](image)

### 9.2.4 Evaluation

According to the typology described in [4], the Opinion category covers a number of concrete concepts, as follows.

- **problem/diagnosis**: the assignment of a known diagnosis or problem label to a set of observed signs and symptoms in the patient, for the purpose of determining and managing treatment. The physician will usually include a date of initial onset, date clinically recognised, date of last occurrence, date of resolution of last occurrence and possibly other timing information.

- **risk assessment**: an evaluation of the likelihood and timing of a certain event occurring or condition appearing.

- **scenario**: an opinion about the outcome if a certain intervention is proceeded with.

- **goal**: statement of a target, and a time at which it should be reached.
**recommendation**: a description in general terms of a suggested care approach for the patient, based on diagnosis; includes various times or time-periods for activities, such as monitoring, taking of medications, and review.

The approach taken to modelling these concepts in openEHR is heavily based upon the development of archetypes for assessments such as “diagnosis” (various kinds), “goal”, “adverse reactions”, “alert”, “exclusion”, “clinical synopsis”, “risk based on family history” and so on. Experience has shown that the Opinion category is too variable for safely modelling its sub-categories directly in the reference model. Instead, a single class `EVALUATION` is used for all instances of the Opinion category. (The name Evaluation has been present in openEHR for some years, and is retained for reasons of continuity).

The design of the `EVALUATION` class is very simple. In addition to the attributes inherited from `ENTRY` and `CARE_ENTRY`, it has only one attribute, `data`: `ITEM_STRUCTURE`. This structure is intended to be archetyped so as to model all the details of any particular clinical information in the Opinion category. No timing attributes are included, since there is no time associated with creation or capturing of Evaluation information as such, only times included in the information. The only times of generic significance are (potentially) the time of a patient consultation during which the Evaluation was created (recorded in `COMPOSITION.event_context.start_time` and `end_time`) and the time of committal to the EHR system (recorded in `VERSION.commit_audit` attribute).

The general meaning of the inherited attributes is as for all Entries. In Evaluations, the `provider` is almost always the physician, while the `protocol` may be used to indicate how a particular assessment was made. The `other_participations` attribute is not as likely to be used for Evaluations representing diagnoses, since a diagnosis is usually the result of thinking on the part of the physician; an exception to this would be a case conference or if an expert system were used. However, Plans for complex patients may well be constructed by multiple physicians.

### 9.2.5 Instruction and Action

Instructions in openEHR specify actions to be performed in the future. They differ from information in the Proposal sub-category of Opinion in the typology (i.e. instances of Evaluation in the class model) in that they are specified in sufficient detail to be directly enacted without further clinical decision-making related to the design of the Instruction, e.g. they can be performed by the patient or a nurse. Any decisions that could be made during the performance of an Instruction are either constrained by the Instruction itself (e.g. dose range; suspend if adverse reaction) or else are assumed knowledge of the expected performer. For example, an Evaluation may say that “oral cortico-steroids are indicated at a peak flow of 200 l/m”. A corresponding Instruction would indicate the actual drug, route, dose, frequency, and so on. The informed patient might be reasonably expected to be able to vary the dosage on his or her own within a dosage guideline explained by his/her GP.

In the typology of FIGURE 19, Instructions are further categorised as Investigation and Intervention. However, as for Evaluation, only a single key class, `INSTRUCTION`, is used to model all types of the Instruction category, with archetypes defining the details of the Instruction. A second Entry subtype, `ACTION`, is used to model the information recorded due to the execution of an Instruction by some agent.

The following subsections describe Instruction and Action in some detail.

#### 9.2.5.1 Requirements

The Instruction and Action classes are designed to satisfy the following requirements:
All kinds of interventions, from simple medication orders to complex multi-drug courses should be representable using the same model;

Instructions should always have a narrative expression, with an optional machine-processible expression in cases where automation will be used;

The freedom must exist to model any particular intervention in as much or little detail as required by circumstances;

Clinicians must be able to specify Instruction steps in their own terms, i.e. using terms like “prescribe”, “dispense”, “start administering”, etc;

Instructions representing diverse clinical workflows must be queryable in a standard way, so that it can be ascertained what Instructions are ‘active’, ‘completed’ and so on for a patient;

It should be possible to provide a coherent view of the state of execution of an Instruction even if parts of it have been executed in different healthcare provider environments;

It must be possible to record ad hoc actions in the record, i.e. acts for which no Instruction was defined (at least in the EHR in question);

Instructions must integrate with notification / alert services;

An interoperable expression of computable workflow definitions of Instructions will be supported.

9.2.5.2 Design Principles

The design approach is based on four principles. The first is that the specification of an Instruction is distinguished from the information representing actions performed as a result. This makes the model and resulting information instances clear to software designers and data users. It also enables workflow engines to determine which parts of the specification have already been executed, and allows for Actions actually performed to differ from those specified. The separation is realised in terms of the INSTRUCTION and ACTION classes and their helpers. Instances of the former specify an Instruction, while instances of the latter describe steps which have actually been performed.

The second principle is the use of a standard instruction state machine (ISM) defining possible states and transitions for any Instruction, no matter what it is. The use of standardised states means that the execution state of any given Instruction can be characterised in exactly the same way (e.g. ‘planned’, ‘active’, etc), and that it is therefore possible to query the EHR and find out all interventions of any kind in a particular state.

The third principle is to provide a way of mapping steps in any care pathway (i.e. healthcare business process) to states in the Instruction state machine. A care pathway is a process which covers the entirety of steps required to effect an Instruction, including prescribing, booking, dispensing, referring, suspension etc. Any such step when performed leaves the Instruction in one of the states of the ISM.

The fourth principle is to support the expression of the formal workflow definition for an Instruction, where full automation is required. It must be recognised that automation of most therapies and drug administration, as well as other interventions like biopsies is minimal today, and is likely to remain so for some time. This is for the simple reason that the cost of automating most tasks is prohibitive compared to human execution, particularly when Instruction activities can often be executed by healthcare professionals already present for other reasons (e.g. ward nurses). It also has to be said that serious research into the use of workflow automation in healthcare is only quite recent, and that so far, there are no standard models for clinical workflow. In the openEHR approach to modelling workflow, such uncertainty is dealt with in two ways. Firstly, formal workflow specification of an Instruction is an optional addition to the base model of Instruction and Action classes, and is not required to obtain
a basic level of computability, including use of the ISM. Secondly, the formal expression of workflow is in the form of parsable syntax rather than objects. This is a generally appropriate design choice, since the safest and most convenient persistent form of of a complex formalism is the syntax form rather than the parsed fine-grained object form; this both optimises storage and allows for changes in the syntax over time.

9.2.5.3 Model Overview

Instruction definitions are modelled in terms of the INSTRUCTION and ACTIVITY classes, with optional workflow attributes. These two classes carry the basic information relating to an Instruction, with all formal workflow definition expressed in parsable syntax in the INSTRUCTION.wf_definition attribute. An INSTRUCTION instance includes the narrative description of the Instruction, and a list of ACTIVITY instances. It also includes all the attributes inherited from the CARE_ENTRY class, including subject, participations and so on.

Most Instructions will have only one Activity, usually describing a medication to be administered and its timing. Some will have more than one drug or therapy, such as the typical 3 drug Losec-HP regime for treating ulcers, and multi-drug chemotherapy. The base Instruction model does not explicitly try to indicate the exact order, serial or parallel administration, or other dependencies, since the knowledge of how to administer the drugs is known by the relevant clinicians, and/or contained in published guidelines. However, the timing information in each Activity does indicate times, days and the usual specifications of “with meals” etc. The timing information is also sufficient to specify a three drug chemotherapy regime, by indicating which days each drug is administered on. It is only when the Instruction is to be automated by a workflow engine that the full structure of the Activity graph will be given. Activity instances may be completely absent from an Instruction, in which case only the narrative will be present. This will typically occur with imported legacy data which itself has no structured representation of medications.

Overall, there are three levels of representation of an Instruction, as shown in FIGURE 23. It is expected that the vast majority of openEHR systems for the foreseeable future will support only the minimal and basic levels.

![FIGURE 23 Levels of Instruction representation](image-url)

When some part of an Instruction is executed (or in some cases where there is no Instruction), an ACTION instance is created describing what was done, and committed to the EHR. Actions include the inherited CARE_ENTRY attributes, along with the time of being performed, a description of what was performed, and an ISM_DETAILS object indicating which state of the ISM the Instruction is now in. If the Action did correspond to an Instruction, there will be an INSTRUCTION_DETAILS instance, which indicates which Activity of which Instruction was executed, including workflow execution.
details if relevant. Actions for which there is no Instruction (or at least not in the current EHR) do occur fairly often, and will not have the INSTRUCTIONDETAILS part of the information. However, they do have the ISMDETAILS instance, and it should always be possible for the committer of an ad hoc ACTION to determine what state in the ISM it corresponds to (e.g. to at least be able to say that it is active, suspended, completed etc). In this way, the ISM state of all things happening to the patient can be ascertained by querying, regardless of whether explicit Instructions exist.

9.2.5.4 The Standard Instruction State Machine (ISM)

When a healthcare process is unfolding in a clinical context, EHR users want to know things such as the following:

- what is the current step in the process for the patient?
- for a given patient, what processes are planned, active, suspended, completed?
- for the populations of patients, what is the state of a particular workflow, such as a recall?

The approach chosen here to support such functionality is to define a standard instruction state machine whose state transitions can be mapped to the steps of a specific care pathway, enabling it to be used as a descriptive device for indicating the state of any Instruction. The state machine is illustrated in FIGURE 24. This state machine is the result of long term experience with clinical workflows and act management systems. The states are as follows1.

INITIAL: initial state, prior to planning activity (default starting state for computable representation of state machine).

PLANNED: the action has been described, but has not as yet taken place.

POSTPONED: the action has not taken place and will not without specific conditions being met. Specifically, events and conditions that would normally ‘activate’ the instruction will be ignored, until a restore event occurs.

SCHEDULED: the action will take place at some designated future time, and has been booked in a scheduling system.

CANCELLED: the action was defined, but was cancelled before anything happened; it has not and will not take place.

ACTIVE: the action is taking place according to its definition. The entire course of medication or therapy corresponds to this state.

SUSPENDED: the action was begun, but has been stopped temporarily, and will not be restarted until explicitly resumed.

ABORTED: the action began but was permanently terminated before normal completion.

COMPLETED: the action began and was completed normally.

EXPIRED: the time during which the action could have been relevant has expired; the action may have completed, been cancelled, or never occurred.

States CANCELLED, ABORTED and COMPLETED are all terminal states. The EXPIRED state is a pseudo-terminal state, from which transitions are allowed to proceed to any of the true terminal states, due to information being received after the fact (such as a patient reporting that they did indeed finish a course of antibiotics). However it is likely in the EHR that Instructions for many simple medications will finish in the EXPIRED state and remain there.

1. The SCHEDULED state was inspired by Van de Velde & Degoulet, Fig 5.5 [15]
FIGURE 24 openEHR standard Instruction State Machine
The transitions are self-explanatory for the most part, however a few deserve comment. The start and finish events correspond to situations when the administration is not instantaneous, as is the situation with most medications. The do event is equivalent to the finish event occurring immediately after the start event, corresponding to an instantaneous administration, completion of which puts the whole Instruction in the completed state. A single shot vaccination or patient taking a single tablet are typical examples. The states PLANNED, POSTPONED, ACTIVE, SUSPENDED, each have a xxx_step transition which return the state machine to the same state. Workflow steps that cause no transition are mapped to these events and thus leave the Instruction in the same state. An example is a medication review, which will leave the medication in the ACTIVE state, if the physician chooses to continue.

In the future, if delegation of an Instruction to another Instruction is required, nesting of a new Instruction state machine within the Active state of a previous one might need to be supported.

The state machine states and transition names are defined in the openEHR terminology groups “ISM states” and “ISM transitions” respectively.

9.2.5.5 Careflow Process to State Machine Mapping

From a health professional’s point of view, a healthcare workflow, or “careflow”, consists of steps and events designed to meet one or more goals. The steps are highly dependent on the particular kind of workflow, and will usually be named in terms familiar to the relevant kinds of clinical professionals, such as “prescribe”, “book”, “suspend” and so on (note that some of these names may be the same as ISM transitions, but may or may not indicate the same thing). However, the need of users of health information is to know what state the execution of an Instruction is in, regardless of which particular careflow step might have just been executed. This is achieved by defining the mapping between the steps of a particular careflow to the states of the ISM in the archetype for the Instruction. When each Action corresponding to a particular Instruction is performed, it will be known both which careflow step it corresponds to and which ISM state the Instruction is now in. The following table provides an example of the mapping for a UK GP medication order workflow.

<table>
<thead>
<tr>
<th>UK GP Workflow Step</th>
<th>State machine transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>initiate (initial -&gt; planned)</td>
</tr>
<tr>
<td>Prescribe</td>
<td>planned_step (planned -&gt; planned)</td>
</tr>
<tr>
<td>Dispense</td>
<td>start (planned -&gt; active)</td>
</tr>
<tr>
<td>Administer</td>
<td>active_step (active -&gt; active)</td>
</tr>
<tr>
<td>Request Renewal</td>
<td>active_step (active -&gt; active)</td>
</tr>
<tr>
<td>Re-issue</td>
<td>active_step (active -&gt; active)</td>
</tr>
<tr>
<td>Review</td>
<td>active_step (active -&gt; active)</td>
</tr>
<tr>
<td>Finish</td>
<td>finish (active -&gt; completed)</td>
</tr>
<tr>
<td>Cancel</td>
<td>abort (active -&gt; aborted)</td>
</tr>
</tbody>
</table>

Mappings like this are specified in the archetypes for the Instruction. When an ACTION instance is committed to the EHR, the ISM_DETAILS object records the step performed and the ISM state and transition. The careflow step must be one of the steps from the corresponding Instruction. See Relationship to Archetypes on page 64 for details of how archetypes are used to represent such mappings.

9.2.5.6 Clinical Workflow Definition

Clinical workflows exist at multiple hierarchical levels, from the health system level (reduce diabetes costs; manage obesity) to the fine-grained (asthmatic medication prescription for a particular patient). At all levels, there are goals, actors and tasks designed to satisfy the goals. At a coarse-grained busi-
ness process level, workflows may be enacted by more than one actor, and may encompass the whole cycle of Observation, Diagnosis, Instruction and Action. For example a workflow describing the steps “prescribe”, “dispense”, “administer”, “repeat”, “review” and so on, around a medication order might include GP, pharmacist, patient as actors. At the finer level of an actual drug or therapy administration, there is usually a single agent or group that performs a specific task, usually within one provider institution, or at home. The correspondence between workflows at these different levels and particular patients, rather than just categories of patients (e.g. all insulin-dependent diabetics), typically increases at finer levels of granularity. Thus from the point of view of automation, it is likely to be fine-grained workflows that have patient-specific definitions which would reasonably appear in the EHR. Most typical medication administrations are in this category. How automated workflow definitions at higher levels of organisational hierarchy are represented and coordinated with lower-level automated workflows is known to be a difficult problem generally, and given that health computing is generally more complex than most domains, implementation of distributed, coordinated (or “orchestrated” or “choreographed”, to use the terms of the workflow community) clinical workflows is likely to be some years off.

In this context, the possible scope for formal workflow definition in openEHR appears to be as follows.

- To enable the recording of links between openEHR Entries and workflow executions, e.g. a particular guideline. This allows openEHR data to be integrated with coarse-grained non-patient specific workflows.
- To support a standardised, interoperable representation of fine-grained formal workflow definitions for activities like medication administration.
- To use formal workflow definitions only where automation is actually useful, and is likely to be used. The kind of workflows that are likely to be worth automating are those which run over several days, weeks or longer, i.e. where humans might easily forget to perform a step. In these cases, the output of the workflow system will be reminders for humans to do certain things at certain times, rather than direct machine automation of the task. Execution of such workflow definitions will generate entries in “worklists” for staff or other agents to perform. Examples include asthma drug management for a child and PAP recall management.
- To ensure that any workflow definition takes account of other existing clinical activities, i.e. does not attempt to define all activities that might possibly be relevant. A simple example is a workflow for asthma medication administration probably does not need to explicitly model the taking of peak flow measurements, since this would normally be occurring anyway.

There are various technical challenges with proposing a standard workflow formalism for clinical use. Firstly, executable workflow definitions are essentially structured programs, similar to programs in procedural languages, but with the addition of temporal logic operators, including alternative paths, parallel paths, wait operations, and also references to outside data sources and services. Recent work in clinical workflow modelling e.g. [9], [1], [7] appears to favour a structural (i.e. parse-tree) approach to representation, due to the need to compute potential modifications to an executing workflow, including dropping, replacing and moving nodes. (Whether such “live” modification of executing workflows is realistic from the designer’s point of view might be questionable, since it means that the design of each workflow has include every possible exceptional case at a detailed level.)

Secondly, the need to connect workflows to the outside world, i.e. data sources and services like notification and worklist management is crucial in making workflows and guidelines implementable.
This problem is probably the main weakness of all guideline and workflow languages to date, including Arden, GLIF, and the various workflow languages such as those mentioned earlier.

The approach taken by the current release of openEHR in representing computable workflow is the following.

- Where used at all, formal workflow definitions are expressed in terms of syntax, not structures, since syntax is always a more appropriate representation for persistence (just as object structures, i.e. parse trees are more appropriate for computation).
- Access to patient data items are expressed within the syntax as symbolic queries.
- Actions, such as requests to the notification service are represented as symbolic commands.

The entire definition of a workflow is expressed as an optional parsable string, in the \textit{wf_definition:DV_PARSABLE} attribute of the \textit{INSTRUCTION} class. Any syntax may currently be used. A workflow syntax is under development by the openEHR Foundation, which is designed to incorporate the relevant features of current workflow models and research, while integrating it into the openEHR type system and archetype framework. In particular, early versions of this syntax will show how patient data access and service commands can be expressed.

\textbf{9.2.5.7 Relationship to Archetypes}

Much of the semantics of particular Instructions and Actions derive from archetypes. Currently, archetypes are used to define two groups of Instruction semantics. The first is the descriptions of activities that are defined in Instructions (\textit{ACTIVITY.description}) and executed in Actions (\textit{ACTION.description}). These descriptions are always of the same form for any given Instruction, and it is highly desirable to have the same archetype component for both. An example is where the description is of a medication, commonly consisting of a tree or list of ten or more elements describing the drug, its name, form, dose, route and so on. The same information structure is needed in the Instruction, where it defines what is to be administered, and in the Action, where it describes what has been administered. In any particular instance, there may be small differences in what was administered (e.g. dose or route are modified) even though the archetype model will be the same for both.

In terms of archetypes therefore, definition of say two \textit{ACTIVITIES} in an \textit{INSTRUCTION} (see example illustrated in FIGURE 25) will actually create separate archetypes of the Activity structures, each of which will be one of the subtypes of \textit{ITEM_STRUCTURE} (since this is the type of \textit{ACTIVITY.description} and \textit{ACTION.description}). The archetypes will then be used by both the \textit{INSTRUCTION} archetype and the \textit{ACTION} archetype, via the archetype slot mechanism (i.e. the standard way of composing archetypes from other archetypes; see \textit{use_archetype} statements in FIGURE 25).

The second category of archetypable semantics is the correspondence between steps in a healthcare business process and the standard instruction state machine, as described above. This mapping is an archetype of the \textit{ism_transition} attribute of an \textit{ACTION} attribute, and therefore defines part of the \textit{ACTION} archetype. FIGURE 25 shows how logical archetype elements in the archetyped editor environment corresponds to the resulting archetypes.
FIGURE 25 Archetypes, Instruction, and Action
9.3 Class Descriptions

9.3.1 ENTRY Class

### Purpose
The abstract parent of all ENTRY subtypes. An ENTRY is the root of a logical item of “hard” clinical information created in the “clinical statement” context, within a clinical session. There can be numerous such contexts in a clinical session. Observations and other Entry types only ever document information captured/created in the event documented by the enclosing Composition.

An ENTRY is also the minimal unit of information any query should return, since a whole ENTRY (including subparts) records spatial structure, timing information, and contextual information, as well as the subject and generator of the information.

### CEN
Cluster OCC (ENV 13606:2000); Entry (prEN 13606:2006)

### OMG HDTF
COAS::HealthRecordEntry and COAS::ObservationQualifier, a generic class which is used to represent context attributes which are concretely modelled here.

### Synapses
The Item class is the closest match for Entry as described here.

### GEHR
* _CONTENT

### HL7v3
Act

### Inherit
CONTENT_ITEM

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>language: CODE_PHRASE</td>
</tr>
<tr>
<td>1..1</td>
<td>encoding: CODE_PHRASE</td>
</tr>
</tbody>
</table>
| 1..1      | subject: PARTY_PROXY | Id of human subject of this ENTRY, e.g.:
  - organ donor
  - foetus
  - a family member
  - another clinically relevant person. |
<table>
<thead>
<tr>
<th>CLASS</th>
<th>ENTRY (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>provider: PARTY_PROXY</td>
<td>Optional identification of provider of the information in this ENTRY, which might be: • the patient • a patient agent, e.g. parent, guardian • the clinician • a device or software Generally only used when the recorder needs to make it explicit. Otherwise, Composition composer and other participants are assumed.</td>
</tr>
<tr>
<td>other_participations: List &lt;PARTICIPATION&gt;</td>
<td>Other participations at ENTRY level.</td>
</tr>
<tr>
<td>workflow_id: OBJECT_REF</td>
<td>Identifier of externally held workflow engine data for this workflow execution, for this subject of care.</td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject_is_self: Boolean</td>
<td>Returns True if this Entry is about the subject of the EHR, in which case the subject attribute is of type PARTY_SELF.</td>
</tr>
</tbody>
</table>

**Invariants**

- Language_valid: language /= Void and then code_set("languages").has(language)
- Encoding_valid: encoding /= Void and then code_set("character sets").has(encoding)
- Subject_validity: subject_is_self implies subject.generating_type = “PARTY_SELF”
- Other_participations_valid: other_participations /= Void implies not other_participations.is_empty
- Archetype_root_point: is_archetype_root

### 9.3.2 ADMIN_ENTRY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ADMIN_ENTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Entry subtype for administrative information, i.e. information about setting up the clinical process, but not itself clinically relevant. Archetypes will define contained information.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for admistrative details of admission, episode, ward location, discharge, appointment (if not stored in a practice management or appointments system).</td>
</tr>
<tr>
<td>Misuse</td>
<td>Not used for any clinically significant information.</td>
</tr>
<tr>
<td>Inherit</td>
<td>ENTRY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

© 2003-2006 The openEHR Foundation  
email: info@openEHR.org web: http://www.openEHR.org
### 9.3.3 CARE_ENTRY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CARE_ENTRY (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The abstract parent of all clinical ENTRY subtypes. A CARE_ENTRY defines protocol and guideline attributes for all clinical Entry subtypes.</td>
</tr>
<tr>
<td>Inherit</td>
<td>ENTRY</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>protocol</td>
<td>ITEM_STRUCTURE</td>
</tr>
<tr>
<td>guideline_id</td>
<td>OBJECT_REF</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>

### 9.3.4 OBSERVATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OBSERVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Entry subtype for all clinical data in the past or present, i.e. which (by the time it is recorded) has already occurred. OBSERVATION data is expressed using the class HISTORY&lt;T&gt;, which guarantees that it is situated in time.</td>
</tr>
<tr>
<td>Use</td>
<td>OBSERVATION is used for all notionally objective (i.e. measured in some way) observations of phenomena, and patient-reported phenomena, e.g. pain.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Not used for recording opinion or future statements of any kind, including instructions, intentions, plans etc.</td>
</tr>
<tr>
<td>CEN</td>
<td>Cluster</td>
</tr>
</tbody>
</table>
### 9.3.5 EVALUATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OBSERVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEHR</td>
<td>G1_OBSERVATION_CONTENT, G1_SUBJECTIVE_CONTENT</td>
</tr>
<tr>
<td>HL7v3</td>
<td>Observation</td>
</tr>
<tr>
<td>Inherit</td>
<td>CARE_ENTRY</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: HISTORY</td>
<td>The data of this observation, in the form of a history of values which may be of any complexity.</td>
</tr>
<tr>
<td>state: HISTORY</td>
<td>Optional recording of the state of subject of this observation during the observation process, in the form of a separate history of values which may be of any complexity. State may also be recorded within the History of the data attribute.</td>
</tr>
</tbody>
</table>

#### Invariants

*Data_valid: data /= Void*

---

### Class EVALUATION

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EVALUATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Entry type for evaluation statements.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for all kinds of statements which evaluate other information, such as interpretations of observations, diagnoses, differential diagnoses, hypotheses, risk assessments, goals and plans.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Should not be used for actionable statements such as medication orders - these are represented using the INSTRUCTION type.</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_SUBJECTIVE_CONTENT</td>
</tr>
<tr>
<td>HL7v3</td>
<td>In HL7v3, diagnoses etc seem to be Observations.</td>
</tr>
<tr>
<td>Inherit</td>
<td>CARE_ENTRY</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: ITEM_STRUCTURE</td>
<td>The data of this evaluation, in the form of a spatial data structure.</td>
</tr>
</tbody>
</table>

#### Invariants

*Data_valid: data /= Void*
### 9.3.6 INSTRUCTION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Used to specify actions in the future. Enables simple and complex specifications to be expressed, including in a fully-computable workflow form.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for any actionable statement such as medication and therapeutic orders, monitoring, recall and review. Enough details must be provided for the specification to be directly executed by an actor, either human or machine.</td>
</tr>
<tr>
<td>Misuse</td>
<td>Not to be used for plan items which are only specified in general terms.</td>
</tr>
</tbody>
</table>

**GEHR**

G1_INSTRUCTION

**HL7v3**

Act subtype Substance_administration, any Act type which is really an action specification (cf an Act in the past)

**Inherit**

CARE_ENTRY

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>narrative: DV_TEXT</td>
<td>Mandatory human-readable version of what the Instruction is about.</td>
</tr>
<tr>
<td>activities: List&lt;ACTIVITY&gt;</td>
<td>List of all activities in Instruction.</td>
</tr>
<tr>
<td>expiry_time: DV_DATE_TIME</td>
<td>Optional expiry date/time to assist determination of when an Instruction can be assumed to have expired. This helps prevent false listing of Instructions as Active when they clearly must have been terminated in some way or other.</td>
</tr>
<tr>
<td>wf_definition: DV_PARSABLE</td>
<td>Optional workflow engine executable expression of the Instruction.</td>
</tr>
</tbody>
</table>

#### Invariants

- **Narrative_valid**: narrative /= Void
- **Activities_valid**: activities /= Void implies not activities.is_empty

### 9.3.7 ACTIVITY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ACTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Defines a single activity within an Instruction, such as a medication administration.</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
### 9.3.8 ACTION Class

**CLASS** ACTION

**Purpose**
Used to record a clinical action that has been performed, which may have been *ad hoc*, or due to the execution of an Activity in an Instruction workflow. Every Action corresponds to a careflow step of some kind or another.

**Inherit**
CARE_ENTRY

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 time: DV_DATE_TIME</td>
<td>Point in time at which this action completed.</td>
</tr>
<tr>
<td>1..1 description: ITEM_STRUCTURE</td>
<td>Description of the activity to be performed, in the form of an archetyped structure.</td>
</tr>
<tr>
<td>1..1 ism_transition: ISM_TRANSITION</td>
<td>Details of transition in the Instruction state machine caused by this Action.</td>
</tr>
<tr>
<td>0..1 instruction_details: INSTRUCTIONDETAILS</td>
<td>Details of the Instruction that caused this Action to be performed, if there was one.</td>
</tr>
</tbody>
</table>

**Invariants**

- **Time_valid**: time /= Void
- **Description_valid**: description /= Void
- **Ism_transition_valid**: ism_transition /= Void

### 9.3.9 INSTRUCTIONDETAILS Class

**CLASS** INSTRUCTIONDETAILS

**Purpose**
Used to record details of the Instruction causing an Action.

**Inherit**
LOCATABLE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 instruction_id: OBJECT_VERSION_ID</td>
<td>Id of causing Instruction.</td>
</tr>
</tbody>
</table>
## 9.3.10 ISM_TRANSITION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>INSTRUCTIONDETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 activity_id: String</td>
<td>Identifier of Activity within Instruction, in the form of its archetype path.</td>
</tr>
</tbody>
</table>
| 0..1 wf_details: ITEM_STRUCTURE | Various workflow engine state details, potentially including such things as:  
• condition that fired to cause this Action to be done (with actual variables substituted);  
• list of notifications which actually occurred (with all variables substituted);  
• other workflow engine state.  
This specification does not currently define the actual structure or semantics of this field. |

**Invariants**

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
</table>
| Instruction_id_valid: instruction_id /= Void  
Activity_path_valid: activity_id /= Void and then not activity_id.is_empty |

### 9.3.10 ISM_TRANSITION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ISM_TRANSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model of a transition in the Instruction State machine, caused by a careflow step. The attributes document the careflow step as well as the ISM transition.</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 current_state:</td>
<td>DV_CODED_TEXT</td>
<td>The ISM current state. Coded by openEHR terminology group “ISM states”.</td>
</tr>
<tr>
<td>0..1 transition:</td>
<td>DV_CODED_TEXT</td>
<td>The ISM transition which occurred to arrive in the current_state. Coded by openEHR terminology group “ISM transitions”.</td>
</tr>
<tr>
<td>0..1 careflow_step:</td>
<td>DV_CODED_TEXT</td>
<td>The step in the careflow process which occurred as part of generating this action, e.g. “dispense”, “start_administration”. This attribute represents the clinical label for the activity, as opposed to current_state which represents the state machine (ISM) computable form. Defined in archetype.</td>
</tr>
</tbody>
</table>

**Invariants**

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
</table>
| Current_state_valid: current_state /= Void and then terminology(“openehr”).codes_for_group_name(“ISM states”, “en”).has(current_state.defining_code)  
Transition_valid: transition /= Void implies terminology(“openehr”).codes_for_group_name(“ISM transitions”, “en”).has(transition.defining_code) |
9.4 Instance Structures

The following subsections illustrate typical Entry instance structures. For guidance on how to best model particular clinical statements, see the archetype part of the openEHR knowledge repository (http://svn.openehr.org/knowledge/project_page.htm).

9.4.1 OBSERVATION

Heartrate Measurement Series

FIGURE 26 illustrates three heartrate measurements over 10 minutes.

FIGURE 26 Periodic series Instance Structure
Blood Pressure with Protocol

FIGURE 27 illustrates a blood pressure observation with protocol.

FIGURE 27 Blood Pressure Measurement Observation

Glucose Tolerance Test

An oral glucose tolerance test takes the following form, although the number and timing of the blood sugar levels may be slightly different in practice:

- challenge: no calories fasting from 12pm to 8am
- datum: BSL - 8am
- challenge: 75 g glucose orally - 8:01 am
- datum: BSL - 9 am
- datum: BSL - 10 am

OGTT is treated as a single clinical concept, and thus requires only one archetype. A typical instance structure is shown in FIGURE 28. In this example, the three blood sugars are represented by EVENTs, with the fasting and glucose challenges being expressed as states on the relevant events.

9.4.2 EVALUATION

Partial Asthma Management Plan

FIGURE 29 illustrates a partial asthma management plan in which monitoring (peak flow) with dependent actions (review and admission to ER) and therapy (bronchodilator) are shown. In a complete plan, symptom monitoring and other medications might be shown. The parts of the plan are
**FIGURE 28** OGTT Instance Structure

linked to the root **EVALUATION** node via the **links**: Set<LINK> attribute inherited from the **LOCATABLE** class.

**FIGURE 29** Partial Asthma Management Plan
9.4.3 INSTRUCTION

Chained Medication Order

Often, a medication order for one drug consists of segments in which one or more of the administration details of route, form, frequency, dose etc is changed. In hospitals, intravenous antibiotics and pain relief drugs may be followed by a tablet form of the same drug to be taken orally. Other examples are common in general practice, such as the following order:

- trade name = Panafcortelone; generic name = Prednisolone; form = tablets; dose = 25mg; route = oral; freq = bd x 3 days; od x 2 days.

FIGURE 30 illustrates the instance structure for this Instruction. Note that the timing attribute of the ACTIVITY instance is shown in human-readable form; in reality it will be a GTS string or similar (see Timing Specification section of openEHR Data Types IM).

Multi-drug Therapy

A common regime for treating duodenal ulcer and related complaints is using Losec with other drugs, such as in the following combination:

- Losec 40 mg od x 4w or until no symptoms
- amoxicillin 500 mg 3td x 7d
- metronidizole 400 mg 3td x 7d
The Instruction for this therapy is illustrated in FIGURE 31.

![FIGURE 31 Multi-drug therapy Instruction]
## A Glossary

### A.1 openEHR Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCA</td>
<td>Health care agent - any doctor, nurse or other recognised staff member, or software or device</td>
</tr>
<tr>
<td>HCF</td>
<td>Health care facility - any place where EHRs are kept</td>
</tr>
<tr>
<td>HCP</td>
<td>Health care professional - any doctor, nurse or other recognised staff member of an HCF</td>
</tr>
</tbody>
</table>

### A.2 Clinical Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Care Pathway</td>
<td>A global care management strategy for a patient, showing management of health problems or issues in a time-based framework, similar to a project management view of an engineering work.</td>
</tr>
<tr>
<td>Contribution</td>
<td></td>
</tr>
<tr>
<td>Episode</td>
<td>A series of clinical events linked in time, such as a hospital admission or a surgical episode.</td>
</tr>
<tr>
<td>Event</td>
<td></td>
</tr>
<tr>
<td>Extract</td>
<td></td>
</tr>
<tr>
<td>Issue</td>
<td>A problem as identified by the patient, e.g. “inability to do exercise due to breathing difficulty”; may be the object of wider health care, e.g. social workers, physiotherapists etc.</td>
</tr>
<tr>
<td>Section</td>
<td></td>
</tr>
<tr>
<td>Problem</td>
<td>A health problem of the patient, as identified by its underlying medical cause, e.g. asthma; the object of medical care.</td>
</tr>
</tbody>
</table>

### A.3 IT Terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET API</td>
<td>Application programmer’s interface - the software interface to a library or module.</td>
</tr>
<tr>
<td>COM</td>
<td>Microsoft’s Component Object Model; designed to enable integration of binary components obeying stated exported interfaces.</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture - an object-oriented middleware architecture enabling the construction of 3-tier systems, in which backend data providers (DBMSs etc) are known only by the services they export to the network. CORBA is an open standard managed by the Object Management Group (OMG).</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed version of Microsoft COM. Similar in its aim to CORBA.</td>
</tr>
<tr>
<td>J2EE</td>
<td></td>
</tr>
<tr>
<td>ODMG-93</td>
<td>A standard for object databases, which includes an object definition language (ODL) for writing schemas, an object query language (OQL) for querying, and several language bindings</td>
</tr>
</tbody>
</table>

Date of Issue: 25 Jan 2006  Page 78 of 83  Editors: {T Beale, S Heard}, {D Kalra, D Lloyd}
B References

B.1 General


2. Berners-Lee T. "Universal Resource Identifiers in WWW". Available at http://www.ietf.org/rfc/rfc2396.txt. This is a World-Wide Web RFC for global identification of resources. In current use on the web, e.g. by Mosaic, Netscape and similar tools. See http://www.w3.org/Addressing for a starting point on URIs.


B.2 European Projects


21 Deliverable 4: GEHR Requirements for Clinical Comprehensiveness. GEHR Project 1992

22 Deliverable 7: Clinical Functional Specifications. GEHR Project 1993

23 Deliverable 8: Ethical and legal Requirements of GEHR Architecture and Systems. GEHR Project 1994

24 Deliverable 19,20,24: GEHR Architecture. GEHR Project 30/6/1995


27 Kalra D. (Editor). Synapses ODP Information Viewpoint. EU Telematics Application Programme, Brussels; 1998; The Synapses Project: Final Deliverable. 10 chapters, 64 pages.

B.3 CEN

28 ENV 13606-1 - Electronic healthcare record communication - Part 1: Extended architecture. CEN/ TC 251 Health Informatics Technical Committee.

29 ENV 13606-2 - Electronic healthcare record communication - Part 2: Domain term list. CEN/ TC 251 Health Informatics Technical Committee.

30 ENV 13606-3 - Electronic healthcare record communication - Part 3: Distribution rules. CEN/ TC 251 Health Informatics Technical Committee.

31 ENV 13606-4 - Electronic Healthcare Record Communication standard Part 4: Messages for the exchange of information. CEN/ TC 251 Health Informatics Technical Committee.
B.4 GEHR Australia


B.5 HL7v3

35 Schadow G, Russler D, Mead C, Case J, McDonald C. HL7 version 3 deliverable: The Unified Service Action Model: Documentation for the clinical area of the HL7 Reference Information Model. (Revision 2.4+).
36 Schadow G, Biron P. HL7 version 3 deliverable: Version 3 Data Types. (DRAFT Revision 1.0).

B.6 OMG

37 CORBAmed document: Person Identification Service. (March 1999). (Authors?)
39 CORBAmed document: Lexicon Query Service. (March 1999)

B.7 Software Engineering

40 Meyer B. Object-oriented Software Construction, 2nd Ed.
Prentice Hall 1997
41 Walden K, Nerson J. Seamless Object-oriented Software Architecture.
Prentice Hall 1994
42 Gamma E, Helm R, Johnson R, Vlissides J. Design patterns of Reusable Object-oriented Software
Addison-Wesley 1995
43 Fowler M. Analysis Patterns: Reusable Object Models
Addison Wesley 1997
44 Fowler M, Scott K. UML Distilled (2nd Ed.)
Addison Wesley Longman 2000
45 Booch G, Rumbaugh J, Jacobsen I. The Unified Modelling Language User Guide. Addison es-
B.8 Resources

49 EON ref required
51 IANA - http://www.iana.org/.
53 SynEx project, UCL. http://www.chime.ucl.ac.uk/HealthI/SynEx/.
END OF DOCUMENT
REFERENCE MODEL

The openEHR Demographic Information Model

Editors: {T Beale, S Heard}, {D Kalra, D Lloyd}

Revision: 2.0
Pages: 25

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.

2. You may read and print the document for private, non-commercial use.

3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.

4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).

5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"

6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.

7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>H Frankel, T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T Beale, H Frankel</td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.96</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.7</td>
<td>CR-000133. Remove details /= Void invariant from PARTY</td>
<td>R Chen</td>
<td>12 Mar 2005</td>
</tr>
<tr>
<td>1.4.6</td>
<td>CR-000048. Pre-release review of documents. Corrected STRUCTURE to be ITEM_STRUCTURE. Make ACTOR_languages a List not a Set.</td>
<td>D Lloyd</td>
<td>22 Feb 2005</td>
</tr>
<tr>
<td>1.4.5</td>
<td>CR-000024. Revert meaning to STRING and rename as archetype_node_id. CR-000118. Make package names lower case.</td>
<td>S Heard, T Beale</td>
<td>10 Jan 2005</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T Beale, T Beale</td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.9</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.3</td>
<td>CR-000013. Rename key classes, according to CEN ENV 13606.</td>
<td>S Heard, D Kalra, T Beale</td>
<td>15 Sep 2003</td>
</tr>
<tr>
<td>1.4.2</td>
<td>CR-000035. Clarify circular relationships between PARTY and PARTY_REL.</td>
<td>Z Tun</td>
<td>14 Aug 2003</td>
</tr>
<tr>
<td>1.4.1</td>
<td>CR-000003. Removed ARCHETYPED and VERSIONABLE classes.</td>
<td>T Beale, Z Tun</td>
<td>18 Mar 2003</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Review by H Frankel, MCA. Corrections to diagrams and class texts. Improved PARTY_RELATIONSHIP semantics. Added Patient instance example. Made time_validity attributes optional.</td>
<td>T Beale</td>
<td>13 Feb 2003</td>
</tr>
<tr>
<td>1.3</td>
<td>Corrections to diagrams and class texts. Inheritance changed to ARCHETYPED for most key classes. Some instance examples added.</td>
<td>Z Tun, T Beale</td>
<td>08 Jan 2003</td>
</tr>
<tr>
<td>1.2</td>
<td>General modifications, addition of CAPABILITY class.</td>
<td>T Beale, D Lloyd</td>
<td>22 Oct 2002</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>1.1</td>
<td>Renamed CONTACT_DESCRIPTOR to CONTACT. Removed CONTACT.role. Renamed PARTY_ROLE to ROLE. Changed CONTACT.address to addresses. Renamed SPATIAL to STRUCTURE. Introduced PARTY and ACTOR classes.</td>
<td>T Beale</td>
<td>18 Sep 2002</td>
</tr>
<tr>
<td>1.0</td>
<td>Created from EHR RM.</td>
<td>T Beale</td>
<td>28 Aug 2002</td>
</tr>
</tbody>
</table>

Acknowledgements

The work reported in this paper has been funded in by a number of organisations, including The University College, London; The Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia; Ocean Informatics, Australia.
# Table of Contents

Copyright Notice ....................................................................................... 2  
Amendment Record ...................................................................................... 3  
Acknowledgements ....................................................................................... 4  
Table of Contents ........................................................................................ 5  

1 Introduction ......................................................................................... 7  
1.1 Purpose .............................................................................................. 7  
1.2 Related Documents .......................................................................... 7  
1.3 Status ................................................................................................. 7  
1.4 Peer review ........................................................................................ 7  
1.5 Conformance ....................................................................................... 8  

2 Demographic Package ........................................................................... 9  
2.1 Overview ........................................................................................... 9  
2.1.1 Archetyping .................................................................................. 10  
2.1.2 Names and Addresses ................................................................. 10  
2.1.3 Party Identification ...................................................................... 10  
2.1.4 Party Relationships ...................................................................... 10  
2.1.5 Versioning Semantics .................................................................. 11  
2.2 Class Definitions .............................................................................. 11  
2.2.1 PARTY Class ............................................................................... 11  
2.2.2 PARTY_IDENTITY Class ............................................................. 12  
2.2.3 CONTACT Class .......................................................................... 13  
2.2.4 ADDRESS Class .......................................................................... 13  
2.2.5 ACTOR Class ............................................................................... 14  
2.2.6 PERSON Class .............................................................................. 14  
2.2.7 ORGANISATION Class ................................................................. 15  
2.2.8 GROUP Class ............................................................................... 15  
2.2.9 AGENT Class ............................................................................... 16  
2.2.10 ROLE Class ................................................................................ 16  
2.2.11 CAPABILITY Class ..................................................................... 17  
2.2.12 PARTY_RELATIONSHIP Class ................................................. 17  
2.3 Instance Examples ............................................................................ 18  
2.3.1 Parties .......................................................................................... 18  
2.3.1.1 Person ................................................................................................. 18  
2.3.1.2 Clinician ............................................................................................ 19  
2.3.1.3 Health Care Facility ................................................................. 19  
2.3.1.4 Group ............................................................................................... 19  
2.3.2 Relationships ............................................................................... 21  
2.3.2.1 Familial Relationship ................................................................. 21  
2.3.2.2 Employment Relationship ..................................................... 21  
2.3.2.3 Patient ............................................................................................ 21  

A References ......................................................................................... 23  
A.1 General .......................................................................................... 23  
A.2 CEN ............................................................................................... 23  
A.3 GEHR Australia ............................................................................. 23  
A.4 OMG ............................................................................................. 23  
A.5 HL7 ............................................................................................... 24  
A.6 Software Engineering ................................................................. 24
A.7 Resources.................................................................................................................. 24
1 Introduction

1.1 Purpose
This document describes the architecture of the openEHR Demographic Information Model. The semantics are drawn from previous work in GEHR, existing models in CEN 13606 and the HL7v3 RIM, and other work done in Australia.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development groups using openEHR;
- Academic groups using openEHR;
- 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 openEHR Modelling Guide
- The openEHR Support Information Model
- The openEHR Data Types Information Model
- The openEHR Common Information Model

Other documents describing related models, include:

- The openEHR EHR Information Model
- The openEHR Demographic Model

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

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

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

1.4 Peer review
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.
1.5 Conformance

Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Demographic Package

2.1 Overview

The demographic model illustrated in FIGURE 1 is a generalised model of the facts one might expect to see in a demographic server. The purpose of the model is as a specification of a demographic service, either standalone, or a “wrapper” service for an existing system such as a patient master index (PMI). In the latter situation, it is used to add the required openEHR semantics, particularly versioning, to an existing service.

The general design is based on the scheme of party and accountability described by Fowler [19], and is influenced by clinical adaptations including work done in Australia [11] and the HL7v3 RIM [15] (itself influenced by the Fowler models).

One of the main design criteria of the model is that it expresses attributes and relationships of demographic entities which exist regardless of particular clinical involvements or participations in particular events. Participations are meaningful only within the context of the health record or other relevant model where they record context-specific relationships between demographic entities and events in the real world.

FIGURE 1 rm.demographic Package

The general design is based on the scheme of party and accountability described by Fowler [19], and is influenced by clinical adaptations including work done in Australia [11] and the HL7v3 RIM [15] (itself influenced by the Fowler models).

One of the main design criteria of the model is that it expresses attributes and relationships of demographic entities which exist regardless of particular clinical involvements or participations in particular events. Participations are meaningful only within the context of the health record or other relevant model where they record context-specific relationships between demographic entities and events in the real world.
Another criterion is that instances of the classes in the model must be serialisable into an EHR Extract in an unambiguous way. This requires that each PARTY be a self-contained hierarchy of data, in the same way as distinct COMPOSITIONs in the EHR model are distinct hierarchies in an Extract. In order to ensure this condition, PARTY RELATIONSHIPS must be implemented correctly, so as to prevent endless traversal of all PARTY objects through their relationships, when serialising. See Party Relationships below for details.

2.1.1 Archetyping
The model is designed to be used with archetypes, hence the generic nature of all entities. Every class containing an attribute of the form details:STRUCTURE is a completely archetypable structure. As a result, archetypes can be defined for concepts such as particular kinds of PERSON, ORGANISATION; for actual ROLES such as “health care practitioner”, and for party identities and addresses.

2.1.2 Names and Addresses
Classes have been included for PARTY_IDENTITY and ADDRESS, even though they contain only a link to details, in the form of the generic STRUCTURE class. This is not strictly necessary - it could have been done simply using appropriately named attributes in the classes PARTY and CONTACT - but is done to provide a place to add specific semantics in future releases of the model. It is also expected to make software development easier, since it provides explicit classes to which behaviour and other implementation attributes can be added. Lastly, it allows the notions of PARTY_IDENTITY and ADDRESS to be explicitly used in archetype-authoring tools.

Instances of PARTY_IDENTITY, linked to PARTY by the attribute identities are intended to express the names of people, organisations, and other actors - that is names which are “owned” by the party, e.g. self-declared (in the case of institutions and companies) or by virtue of social relations (names given by parents, tribes etc). Identifiers of Parties given by other organisations, or the state are not represented in this way, and should be recorded in the PARTY.details structure instead (see below).

2.1.3 Party Identification
Identifiers of Parties given by organisations or the state are treated as any other attribute of a Party, i.e. recorded as part of the data in the PARTY.details structure. Identifiers of Party instances in the system are provided in the same way as identifiers of Compositions in the EHR: via the uid attribute (type OBJECT_VERSION_ID) of the containing VERSION. These identifiers are used in all cross-references between Parties, and between Parties and Party_relationships.

2.1.4 Party Relationships
Relationships between parties in the real world may be expressed using PARTY_RELATIONSHIP objects, as illustrated in FIGURE 2.

![FIGURE 2 General Relationship Model](image)

Relationships are considered directional, hence the use of the attribute names source and target, however, these names are otherwise neutral, and give no indication as to the meaning of the relationships, such as which party is responsible and which accountable (for comparison, see the demographic mod-
els of Fowler [19]). Accordingly, each Party involved in a relationship includes it in its relationships list, if it is at the source end, or in the reverse_relationships list, if at the target end.

The usual way to determine the directionality of a relationship between two Parties is usually by which Party’s actions caused the relationship to come into being. For example, a relationship representing the concept “patient”, between a health consumer and a health care organisation would have the consumer as source and the organisation as target.

PARTY_RELATIONSHIPs are stored as part of the data of the PARTY designated as the source. This means that the relationships attribute is by value, while reverse_relationships is by references, as are PARTY_RELATIONSHIP.source and target. The actual kind of reference is via the use of OBJECT_REFs containing HIER_OBJECT_IDS to denote the Version container of a Party, rather than OBJECT_VERSION_IDs, which would denote particular versions. Logically this implements the semantic that such relationships in the real world are between continuants, i.e. the real Parties, not just one of their version instances in a demographic system.

2.1.5 Versioning Semantics
The class PARTY and its descendants ACTOR and ROLE are all potentially versioned in a demographic system. A Version of a PARTY includes all the compositional parts, such as identities, contacts, Party relationships of which it is the source. Every Party is stored in its own Version container.

2.2 Class Definitions

2.2.1 PARTY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTY (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Ancestor of all party types, including real world entities and their roles. A party is any entity which can participate in an activity. The name attribute inherited from LOCATABLE is used to indicate the actual type of party (note that the actual names, i.e. identities of parties are indicated in the identities attribute, not the name attribute).</td>
</tr>
<tr>
<td>CEN</td>
<td>healthcare agent</td>
</tr>
<tr>
<td>HL7</td>
<td>Entity</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>identities: Set&lt;PARTY_IDENTITY&gt;</td>
<td>Identities used by the party to identify itself, such as legal name, stage names, aliases, nicknames and so on.</td>
</tr>
<tr>
<td>contacts: Set&lt;CONTACT&gt;</td>
<td>Contacts for this party.</td>
</tr>
<tr>
<td>relationships: Set&lt;PARTY_RELATIONSHIP&gt;</td>
<td>Relationships in which this role takes part as source.</td>
</tr>
</tbody>
</table>
### 2.2.2 PARTY_IDENTITY Class

**Purpose**

An identity “owned” by a PARTY, such as a person name or company name, and which is used by the party to identify itself. Actual structure is archetyped.

**Inherit**

LOCATABLE

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>details: ITEM_STRUCTURE</td>
<td>The value of the identity. This will often taken the form of a parsable string or a small structure of strings.</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose: DV_TEXT</td>
<td>Purpose of identity, e.g. “legal”, “stagename”, “nickname”, “tribal name”, “trading name”. Taken from value of inherited name attribute.</td>
</tr>
</tbody>
</table>
### 2.2.3 CONTACT Class

**CLASS** CONTACT

**Purpose**
Description of a means of contact of a party. Actual structure is archetyped.

**Inherit**
LOCATABLE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_validity:</td>
<td>Valid time interval for this contact descriptor.</td>
</tr>
<tr>
<td>&lt;DV_INTERVAL&gt;</td>
<td></td>
</tr>
<tr>
<td>addresses:</td>
<td>A set of address alternatives for this purpose and time validity.</td>
</tr>
<tr>
<td>List&lt;ADDRESS&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>purpose:</td>
<td>Purpose for which this contact is used, e.g. “mail”, “daytime phone”,</td>
</tr>
<tr>
<td>DV_TEXT</td>
<td>etc. Taken from value of inherited name attribute.</td>
</tr>
</tbody>
</table>

**Invariants**

<table>
<thead>
<tr>
<th>Purpose_valid:</th>
<th>purpose = name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses_exists:</td>
<td>addresses /= Void and then not addresses.empty</td>
</tr>
</tbody>
</table>

### 2.2.4 ADDRESS Class

**CLASS** ADDRESS

**Purpose**
Address of contact, which may be electronic or geographic.

**CEN**
Address (data type)

**HL7**
Address (AD) (data type)

**Inherit**
LOCATABLE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>details:</td>
<td>The details of the address, in the form of a STRUCTURE. This may take</td>
</tr>
<tr>
<td>ITEM_STRUCTURE</td>
<td>the form of a SINGLE_S, whose data item is a parsable string or a list</td>
</tr>
<tr>
<td></td>
<td>or tree of many parts.</td>
</tr>
</tbody>
</table>
### 2.2.5 ACTOR Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functions</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td></td>
<td><strong>type</strong>: DV_TEXT</td>
</tr>
<tr>
<td></td>
<td><strong>as_string</strong>: String</td>
</tr>
<tr>
<td><strong>Invariants</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Type_valid</strong>: type = name</td>
</tr>
<tr>
<td></td>
<td><strong>Details_exists</strong>: details /= Void</td>
</tr>
</tbody>
</table>

### 2.2.6 PERSON Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Generic description of persons. Provides a dedicated type to which Person archetypes can be targeted.</td>
</tr>
</tbody>
</table>

---

© 2003-2005 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
# 2.2.7 ORGANISATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ORGANISATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Generic description of organisations. An organisation is a legally constituted body whose existence (in general) outlives the existence of parties considered to be part of it.</td>
</tr>
<tr>
<td>CEN</td>
<td>healthcare organisation</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_HCF</td>
</tr>
<tr>
<td>HL7</td>
<td>ORGANIZATION</td>
</tr>
<tr>
<td>Inherit</td>
<td>ACTOR</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>

# 2.2.8 GROUP Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>GROUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A group is a real world group of parties which is created by another party, usually an organisation, for some specific purpose. A typical clinical example is that of the specialist care team, e.g. “cardiology team”. The members of the group usually work together.</td>
</tr>
<tr>
<td>Inherit</td>
<td>ACTOR</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
</tbody>
</table>
### 2.2.9 AGENT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>AGENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Generic concept of any kind of agent, including devices, software systems, but not humans or organisations.</td>
</tr>
<tr>
<td>CEN</td>
<td>healthcare software, healthcare device</td>
</tr>
<tr>
<td>HL7</td>
<td>DEVICE</td>
</tr>
<tr>
<td>Inherit</td>
<td>ACTOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

### 2.2.10 ROLE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ROLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Generic description of a role performed by an actor. The role corresponds to a competency of the party. Roles are used to define the responsibilities undertaken by a party for a purpose. Roles should have credentials qualifying the performer to perform the role.</td>
</tr>
<tr>
<td>Use</td>
<td>Roles correspond to concepts like “general practitioner”, “nurse” and so on.</td>
</tr>
<tr>
<td>CEN</td>
<td>healthcare agent in context</td>
</tr>
<tr>
<td>HL7</td>
<td>ROLE</td>
</tr>
<tr>
<td>Inherit</td>
<td>PARTY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>capabilities: List&lt;CAPABILITY&gt;</td>
<td>The capabilities of this role.</td>
</tr>
<tr>
<td>0..1</td>
<td>time_validity: DV_INTERVAL&lt;DV_DATE&gt;</td>
<td>Valid time interval for this role.</td>
</tr>
</tbody>
</table>
### 2.2.11 CAPABILITY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ROLE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1..1</strong></td>
<td><strong>performer</strong>: PARTY_REF</td>
</tr>
</tbody>
</table>
| **Invariants** | **Capabilities_valid**: capabilities /= Void implies not capabilities.empty  
**Performer_exists**: performer /= Void |

**CLASS** **CAPABILITY**

**Purpose**
Capability of a role, such as “ehr modifier”, “health care provider”. Capability should be backed up by credentials.

**Use**

**Inherit**
LOCATABLE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1..1</strong> credentials: ITEM_STRUCTURE</td>
<td>The qualifications of the performer of the role for this capability. This might include professional qualifications and official identifications such as provider numbers etc.</td>
</tr>
</tbody>
</table>
| **0..1** time_validity: DV_INTERVAL
  `<DV_DATE>` | Valid time interval for the credentials of this capability. |

**Invariants**

**Credentials_exists**: credentials /= Void

### 2.2.12 PARTY_RELATIONSHIP Class

**CLASS** **PARTY_RELATIONSHIP**

**Purpose**
Generic description of a relationship between parties.

**HL7**
RELATIONSHIP_LINK

**Inherit**
LOCATABLE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0..1</strong> details: ITEM_STRUCTURE</td>
<td>The detailed description of the relationship</td>
</tr>
</tbody>
</table>
| **0..1** time_validity: DV_INTERVAL
  `<DV_DATE>` | Valid time interval for this relationship. |
| **1..1** source: OBJECT_REF | Source of relationship. |
| **1..1** target: OBJECT_REF | Target of relationship. |
### 2.3 Instance Examples

In the following instance examples, the values of the attributes *uid*, *source*, *target*, and *reverse_relationships* are not meant to be taken as literally valid OBJECT_IDs - for the purposes of clarity, simple integers have been used.

#### 2.3.1 Parties

##### 2.3.1.1 Person

FIGURE 3 illustrates a possible set of instances for a PERSON, with home and work contact information. There are separate archetypes for the PERSON, each ADDRESS, and each PARTY_IDENTITY.
the following figure, “meaning” is the meaning from the value of the archetype_node_id attribute, functionally derived from the archetype local ontology.

2.3.1.2 Clinician

2.3.1.3 Health Care Facility

2.3.1.4 Group

FIGURE 4 illustrates the demographic information for a cardiac surgery team in a hospital. The group includes a head surgeon, anaesthetist, assistant surgeon, and presumably others (not shown). Each of
these members of the team have an employment relationship with the hospital (shown only for one surgeon, in the interests of clarity).

![Diagram](image-url)

**FIGURE 4** Group Demographics

---

© 2003-2005 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
2.3.2 Relationships

2.3.2.1 Familial Relationship

2.3.2.2 Employment Relationship

2.3.2.3 Patient

FIGURE 5 shows a simple way of representing the patient relationship between a person and a health care organisation.

FIGURE 6 shows the same logical relationship, but with both Parties acting through Roles, representing their status as healthcare consumer and healthcare provider respectively. Each of these Roles has associated credentials which document its official nature within the healthcare system.

FIGURE 5 Simple Patient Relationship

FIGURE 6 Patient Relationship with Roles and Credentials
A References

A.1 General


A.2 CEN

7. ENV 13606-1 - Electronic healthcare record communication - Part 1: Extended architecture. CEN/ TC 251 Health Informatics Technical Committee.
8. ENV 13606-2 - Electronic healthcare record communication - Part 2: Domain term list. CEN/ TC 251 Health Informatics Technical Committee.

A.3 GEHR Australia


A.4 OMG

A.5  **HL7**


A.6  **Software Engineering**


18  Gamma E, Helm R, Johnson R, Vlissides J. *Design patterns of Reusable Object-oriented Software*. Addison-Wesley 1995

19  Fowler M. *Analysis Patterns: Reusable Object Models* Addison Wesley 1997

20  Fowler M, Scott K. *UML Distilled (2nd Ed.*) Addison Wesley Longman 2000


A.7  **Resources**


25  EON ref required


28  ProForma language for decision support. [http://www.acl.icnet.uk/lab/proforma.html](http://www.acl.icnet.uk/lab/proforma.html)

29  SynEx project, UCL. [http://www.chime.ucl.ac.uk/HealthI/SynEx/](http://www.chime.ucl.ac.uk/HealthI/SynEx/)
END OF DOCUMENT
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000162: Allow party identifiers when no demographic data.</td>
<td>S Heard, H Frankel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000167: Add AUTHORED_RESOURCE class.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000179: Move AUDIT_DETAILS to generic package; add REVISION_HISTORY.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000182: Rationalise VERSION.lifecycle_state and ATTESTATION.status.</td>
<td>C Ma, D Kalra</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000065: Add Revision History to change control package.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000187: Correct modelling errors in DIRECTORY class and rename.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000163: Add identifiers to FEEDER_AUDIT for originating and gateway systems.</td>
<td>H Frankel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000165: Clarify use of system_id in FEEDER_AUDIT and AUDIT_DETAILS.</td>
<td>H Frankel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000190: Rename VERSION_REPOSITORY to VERSIONED_OBJECT.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000161: Support distributed versioning. Additions to change_control package. Rename REVISION_HISTORY_ITEM.revision to version_id, and change type to OBJECT_VERSION_ID.</td>
<td>H Frankel, T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.96</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6.2</td>
<td>CR-000159. Improve explanation of use of ATTESTATION in change_control package.</td>
<td>T Beale</td>
<td>10 Jun 2005</td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>CR-000108. Minor changes to change_control package.</td>
<td>T Beale, S Heard, Ken Thompson</td>
<td>10 Dec 2004</td>
</tr>
<tr>
<td></td>
<td>CR-000024. Revert meaning to STRING and rename as archetype_node_id.</td>
<td>T Beale, D Lloyd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000099. PARTICIPATION.function type in diagram not in sync with spec.</td>
<td>R Shackel, T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000116. Add PARTICIPATION.function vocabulary and invariant.</td>
<td>D Lloyd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000118. Make package names lower case. Improve presentation of identification section; move some text to data types IM document, basic package.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000111. Move Identification Package to Support</td>
<td>DSTC</td>
<td></td>
</tr>
</tbody>
</table>

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>CR-000080. Remove ARCHETYPED.concept - not needed in data</td>
<td>DSTC</td>
<td>09 Mar 2004</td>
</tr>
<tr>
<td></td>
<td>CR-000081. LINK should be unidirectional.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000083. RELATED_PARTY.party should be optional.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000085. LOCATABLE.synthesised not needed. Add vocabulary for FEEDER_AUDIT.change_type.</td>
<td>T Beale, S Heard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000086. LOCATABLE.presentation not needed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000091. Correct anomalies in use of CODE_PHRASE and DV_CODED_TEXT.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changed PARTICIPATION.mode, changed ATTESTATION.status, RELATED_PARTY.relationship, VERSION_AUDIT.change_type, FEEDER_AUDIT.change_type to DV_CODED_TEXT.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000094. Add lifecycle state attribute to VERSION; correct DV_STATE.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Formally validated using ISE Eiffel 5.4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.12</td>
<td>CR-000071. Allow version ids to be optional in TERMINOLOGY_ID.</td>
<td>T Beale, D Lloyd</td>
<td>25 Feb 2004</td>
</tr>
<tr>
<td></td>
<td>CR-000044. Add reverse ref from VERSION_REPOSITORY&lt;T&gt; to owner object.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000063. ATTESTATION should have a status attribute.</td>
<td>D Kalra, T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000046. Rename COORDINATED_TERM and DV_CODED_TEXT.definition.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.11</td>
<td>CR-000056. References in COMMON.Version classes should be OBJECT_REFS.</td>
<td>T Beale</td>
<td>02 Nov 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000043. Move External package to Common RM and rename to Identification (incorporates CR-000036 - Add HIER_OBJECT_ID class, make OBJECT_ID class abstract.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.7</td>
<td>CR-000013. Rename key classes according to CEN ENV13606.</td>
<td>S Heard, D Kalra, T Beale</td>
<td>15 Sep 2003</td>
</tr>
<tr>
<td>1.4.6</td>
<td>CR-000012. Add presentation attribute to LOCATABLE.</td>
<td>D Kalra</td>
<td>20 Jun 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000027. Move feeder_audit to LOCATABLE to be compatible with CEN 13606 revision. Add new class FEEDER_AUDIT.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4.5</td>
<td>CR-000020. Move VERSION.charset to DV_TEXT, territory to TRANSACTION. Remove VERSION.language.</td>
<td>A Goodchild</td>
<td>10 Jun 2003</td>
</tr>
<tr>
<td>1.4.4</td>
<td>CR-000007. Add RELATED_PARTY class to GENERIC package.</td>
<td>S Heard, D Kalra</td>
<td>11 Apr 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000017. Renamed VERSION.parent_version_id to preceding_version_id.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.4.3 **Major alterations** due to CR-000003, CR-000004. ARCHETYPE class no longer inherits from LOCATABLE, now related by association. Redesign of Change Control package. Document structure improved. (Formally validated)

T Beale, Z Tun 18 Mar 2003

1.4.2 Moved External package to Support RM. Corrected CONTRIBUTION.description to DV_TEXT. Made PARTICIPATION.time optional. (Formally validated).

T Beale 25 Feb 2003

1.4.1 **Formally validated using ISE Eiffel 5.2.** Corrected types of VERSIONABLE.language, charset, territory. Added ARCHETYPE.uid:OBJECT_ID. Renamed ARCHETYPEID.rm_source to rm_originator, and rm_level to rm_concept; added archetype_originator. Rewrote archetype id section. Changed PARTICIPATION.mode to COORDINATED_TERM & fixed invariant.

T Beale, D Kalra 18 Feb 2003

1.4 **Changes post CEN WG meeting Rome Feb 2003.** Changed ARCHETYPE.meaning from STRING to DV_TEXT. Added CONTRIBUTION.name invariant. Removed AUTHORED_VA and ACQUIRED_VA audit types, moved feeder audit to the EHR RM. VERSIONABLE.code_set renamed to charset. Fixed pre/post condition of OBJECT_ID.context_id, added OBJECT_ID.has_context_id. Changed TERMINOLOGY_ID string syntax.

T Beale, D Kalra, D Lloyd 8 Feb 2003

1.3.5 Removed segment from archetype_id; corrected inconsistencies in diagrams and class texts.

Z Tun, T Beale 3 Jan 2003

1.3.4 Removed inheritance from VERSIONABLE to ARCHETYPE.

T Beale 3 Jan 2003

1.3.3 Minor corrections: OBJECT_ID; changed syntax of TERMINOLOGY_ID. Corrected Fig 6.

T Beale 17 Nov 2002

1.3.2 Added Generic Package; added PARTICIPATION and changed and moved ATTESTATION class.

T Beale 8 Nov 2002

1.3.1 Removed EXTERNAL_ID.iso_oid. Remodelled EXTERNAL_ID into new classes - OBJECT_REF and OBJECT_ID. Remodelled all change control classes.

T Beale, D Lloyd, M Darlison, A Goodchild 22 Oct 2002

1.3 Moved ARCHETYPE_ID.iso_oid to EXTERNAL_ID. DV_LINK no longer a data type; renamed to LINK.

T Beale 22 Oct 2002

1.2 Removed Structure package to own document. Improved CM diagrams.

T Beale 11 Oct 2002

1.1 Removed HCA_ID. Included Spatial package from EHR RM. Renamed SPATIAL to STRUCTURE.

T Beale 16 Sep 2002

1.0 Taken from EHR RM.

T Beale 26 Aug 2002

**Acknowledgements**
The work reported in this paper has been funded in by a number of organisations, including The University College, London; The Cooperative Research Centres Program through the Department of the
Prime Minister and Cabinet of the Commonwealth Government of Australia; Ocean Informatics, Australia.
# Table of Contents

**Copyright** Notice ................................................................. 2  
Amendment Record ................................................................. 3  
Acknowledgements ................................................................. 5  
Table of Contents ........................................................................ 7  

1 **Introduction** ........................................................................ 9  
1.1 Purpose.................................................................................... 9  
1.2 Related Documents .............................................................. 9  
1.3 Status..................................................................................... 9  
1.4 Peer review ........................................................................... 9  
1.5 Conformance.......................................................................... 10  

2 **Overview** ........................................................................... 11  

3 **Archetyped Package** ............................................................ 13  
3.1 Overview.................................................................................. 13  
3.1.1 The Class LOCATABLE.......................................................... 13  
3.1.2 Feeder System Audit............................................................. 14  
3.2 Class Descriptions................................................................. 18  
3.2.1 Class LOCATABLE................................................................. 18  
3.2.2 ARCHETYPED Class............................................................. 19  
3.2.3 LINK Class ........................................................................ 20  
3.2.4 FEEDER_AUDIT Class ......................................................... 22  
3.2.5 FEEDER_AUDIT_DETAILS Class.......................................... 23  

4 **Generic Package** .................................................................. 25  
4.1 Overview.................................................................................. 25  
4.2 Design Principles .................................................................... 25  
4.2.1 Referring to Demographic Entities ....................................... 26  
4.2.2 Participation ........................................................................ 26  
4.2.3 Audit Information ............................................................... 27  
4.2.4 Attestation .......................................................................... 27  
4.3 Class Descriptions................................................................. 28  
4.3.1 PARTY_PROXY Class............................................................ 28  
4.3.2 PARTY_SELF Class............................................................... 28  
4.3.3 PARTY_IDENTIFIED Class .................................................. 29  
4.3.4 PARTYRELATED Class........................................................ 29  
4.3.5 PARTICIPATION Class ......................................................... 30  
4.3.6 AUDIT_DETAILS Class ......................................................... 31  
4.3.7 ATTESTATION Class............................................................ 31  
4.3.8 REVISION_HISTORY Class ............................................... 32  
4.3.9 REVISION_HISTORY_ITEM Class ...................................... 33  

5 **Directory Package** ............................................................... 35  
5.1 Overview.................................................................................. 35  
5.1.1 Paths................................................................................... 35  
5.2 Class Descriptions................................................................. 36  
5.2.1 VERSIONED_FOLDER Class............................................... 36  
5.2.2 FOLDER Class ................................................................. 36
The openEHR Common Information Model

6 Change Control Package ................................................................. 37
   6.1 Overview .................................................................................. 37
   6.2 Design Principles ...................................................................... 37
   6.2.1 The Configuration Management Paradigm ......................... 37
   6.2.2 Organisation of the Repository ........................................... 37
   6.2.3 Change Management ............................................................ 37
   6.2.4 Changes in Time ..................................................................... 38
   6.2.5 General Model of a Change-controlled Repository .............. 39
   6.3 Formal Model .......................................................................... 40
   6.3.1 Overview ............................................................................. 40
   6.3.2 Audits ................................................................................ 41
   6.3.3 Version Lifecycle ................................................................. 42
   6.3.4 Attestation .......................................................................... 43
   6.3.5 Semantics of Copying in Distributed Systems ...................... 43
   6.3.6 Distributed Version Identification ...................................... 44
   6.3.7 Semantics of Merging Versions ......................................... 46
   6.3.8 Semantics of Moving Version Containers ......................... 46
   6.3.9 Transaction Semantics of Contributions ............................. 47
   6.4 Class Descriptions ................................................................. 48
   6.4.1 VERSIONED_OBJECT Class ............................................ 48
   6.4.2 VERSION Class ................................................................ 49
   6.4.3 IMPORTED_VERSION Class .......................................... 50
   6.4.4 MERGED_VERSION Class ............................................. 51
   6.4.5 CONTRIBUTION Class ..................................................... 51
7 Resource Package ........................................................................ 53
   7.1 Overview ................................................................................ 53
   7.1.1 Natural Languages and Translation ..................................... 53
   7.1.2 Meta-data ........................................................................... 53
   7.1.3 Revision History ................................................................. 54
   7.2 Class Definitions ..................................................................... 54
   7.2.1 AUTHORED_RESOURCE Class ........................................ 54
   7.2.2 TRANSLATION_DETAILS Class .................................... 55
   7.2.3 RESOURCE_DESCRIPTION Class ................................. 55
   7.2.4 RESOURCE_DESCRIPTION_ITEM Class ....................... 57
A References ..................................................................................... 59
   A.1 General ................................................................................ 59
   A.2 European Projects .............................................................. 59
   A.3 CEN .................................................................................... 59
   A.4 OMG .................................................................................... 59
   A.5 Software Engineering .......................................................... 60
   A.6 Resources ............................................................................ 60
1 Introduction

1.1 Purpose
This document describes the architecture of the openEHR Common Reference Model, which contains concepts used by other openEHR reference models.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development groups using openEHR;
- Academic groups using openEHR;
- 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 openEHR Modelling Guide
- The openEHR Support Information Model
- The openEHR Data Types Information Model
- The openEHR Data Structures Information Model

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

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

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

New versions are announced on openehr-announce@openehr.org.

Blue text indicates sections under active development.

1.4 Peer review
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.
1.5 Conformance

Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Overview

The common Reference Model comprises a number of packages containing abstract concepts and design patterns used in higher level openEHR models. It is illustrated in FIGURE 1.

![Common Model Packages](image)

FIGURE 1 rm.common Package

The archetyped package described here is informed by a number of design principles, centred on the concept of “two-level” modelling. These principles are described in detail in [1].

The generic package contains classes representing concepts which are generic across the domain, mostly to do with referencing demographic entities from within other data including Participation, Party_proxy, Attestation and so on.

The directory package provides a simple reusable abstraction of a versioned folder structure.

The change_control package defines the generalised semantics of changes to a repository, such as an EHR, over time. Each item in such a repository is version controlled to allow the repository as a whole to be properly versioned in time. The semantics described are in response to medico-legal requirements defined in GEHR [9], and in the ISO Technical Specification 18308 [4]. Both of these requirements specifications mention specifically the version control of the health record.

The resource package defines semantics of an online authored resource, such as a document, and supports multiple language translations, descriptive meta-data and revision history.
## 3 Archetyped Package

### 3.1 Overview

The archetyped package includes the core types LOCATABLE, ARCHETYPED, and LINK. It is illustrated in FIGURE 2.

![Figure 2: rm.common.archetyped Package](image)

#### 3.1.1 The Class LOCATABLE

Every structural class in the openEHR reference model inherits from the LOCATABLE class, ensuring it has both a runtime name, and an archetype_node_id. The archetype_node_id is the standardised semantic code for a node and comes from the corresponding node in the archetype used to create the data. The only exception is at archetype root points in data, where archetype_node_id carries the archetype identifier in string form rather than an interior node id. LOCATABLE also provides the attribute archetype_details, which is non-Void for archetype root points in data, and carries meta-data relevant to root points. The name attribute carries a name created at runtime. The ‘meaning’ of any node is derived formally from the archetype by obtaining the “text” value for the archetype_node_id code from the archetype ontology, in the language required.

The name and archetype_node_id values are often the same semantically, but may differ. For example, in “problem/SOAP” Sections (i.e. headings), the name of a section at the problem level might be “diabetes”, but its meaning will be “problem”. The default value for name should be assumed to be the text value in the local language for the archetype_node_id code on the node in question, unless explicitly set otherwise.

LOCATABLE objects may also have a uid. In the current openEHR architecture, uids are not needed to identify data nodes, since paths are used to reference all nodes inside top-level structures (i.e. COMPOSITIONs etc). The uids required in the “repository” classes, including EHR, VERSIONED_OBJECT etc are defined separately, and are not inherited from LOCATABLE. If uids are used in data for some reason, they can only be meaningfully used at archetype root points, since there is no point in uniquely
identifying sub-nodes of individual archetyped structures - because it makes no sense to reference them rather than the relevant root node. In general, LOCATABLE.uid should be avoided in normal implementations, although it may find some use in EHR Extracts.

The parent feature in LOCATABLE ensures that any LOCATABLE can reference its parent in the compositional hierarchy, and may be implemented in any way convenient.

3.1.2 Feeder System Audit

The data in any part of the EHR may be obtained from a feeder system, i.e. a source system which does not obey the versioning, auditing and content semantics of openEHR (data in the EHR which have been sourced from another openEHR system are dealt with in the Common IM, Change control section). The FEEDER_AUDIT class defines the semantics of an audit trail which is constructed to describe the origin of data which have been transformed into openEHR form and committed to the system. There are a number of important aspects to the problem of transforming data for committal into an openEHR system, dealt with in the following subsections.

Requirements

The model of Feeder audit is designed to satisfy the following requirements with respect to EHR content sourced from non-openEHR systems:

- record medico-legal audit information from the originating system (e.g. pathology lab system) similar to that captured in the AUDITDETAILS class in the change_control package;
- record information identifying the system from which the content was obtained (might not be the originating system);
- record sufficient information to distinguish incoming items from each other, and to enable the detection of duplicates and new versions of the same item.

Design Principles

The design of the Feeder audit part of the reference model is based on a generalised model of data communication in which various elements are identified, as follows:

- the originating system: the computer system where the information item was initially created, e.g. the system at a pathology laboratory or a reporting system for a number of laboratories;
- intermediate systems: any system which moves information from the originating system to openEHR system;
- the feeder system: the intermediate system from which the information item was directly obtained by the openEHR system; this might be the originating system, or it may be a distinct intermediate system;
- the committing openEHR system: the openEHR system where the information item is transformed into openEHR form and committed as a Composition;
- openEHR converter: a component whose job it is to convert non-openEHR information into a form compliant with the openEHR reference model and chosen archetypes.

FIGURE 3 illustrates these elements, shown as a “feeder chain”, along with typical meta-data available in messages from each system. In general, not much can be assumed about systems in the feeder chain. The originating system may or may not correspond to the place of the clinical activity - it is not uncommon for a pathology company to have a centralised report issuing location while having numerous physical laboratories. There is often limited consistency in the way identifiers are assigned, timestamps are created, and information is structured and coded. In general, information from a
feeder system is in response to a request, often a pathology order, although the request/response pattern probably cannot be assumed in all cases.

The idea underlying the openEHR Feeder audit model is that there are two groups of meta-data which should be recorded about an imported information item. The first is medico-legal meta-data about its creation: the system of origin, who created it and when it was created. The second is identifying meta-data for the item from the originating and feeder system, and potentially other intermediate systems in the feeder chain, where necessary to support duplicate detection, version detection and so on.

Meta-data
The potentially available medico-legal meta-data about the received item is as follows:

- identifier of the originating system (where the item was originally committed);
- identifier of the information item in the originating system;
- agent who committed the item;
- timestamp of committal or creation of the item;
- type of change, e.g. initial creation, correction (including deletion of a subpart), logical deletion (e.g. due to cancellation of order);
- status of information, e.g. interim, final;
- version id, where versioning is supported.

The above information is equivalent to the audit trail and versioning data captured when information created in an openEHR system is committed in a Composition version.

Various kinds of identifying information may be required including the following:

- subject identifier (often more than one, e.g. national patient id, GP’s local patient id, lab’s local patient id) are usually recorded and may be required for traceability purposes;
- subject identifier(s) may identify someone other than the subject of the record as being the subject of the incoming item;
- location of the feeder system;
- identifier of the feeder system (which may be one of many at the feeder system location);
- identifier the feeder system uses for the item in question (often known as an “accession id”);
- identifier of request or order to which the information is a response (sometimes known as a “placer’s request id”);
• identifier of the information item used by the originating system (sometimes known as a “filler’s request id”);
• timestamp(s) assigned by feeder system to the item.

Some or all of this information will usually be sufficient to perform a number of tasks as follows.

**Traceability**

The first is to support medico-legal investigation into the path of the information item through the health computing infrastructure. This requires the availability of sufficient identifier information that the origin of the information item can be traced.

Subject identifiers where available should be used to ensure that the received data go into the correct EHR, by ensuring that the relevant lookups in client directories or other lookup mechanisms can be effected. Again, in rare cases, the subject of the incoming data item may not necessarily be the subject the EHR - a test result may be made from a relative or other associate which will be stored in the patient’s EHR.

**Version Detection**

The second is to detect new versions of an item (e.g. interim and final versions of a microbiology test result). This can usually be achieved by using various identifiers as well as the originating system version id and/or content status (interim, final etc). A new openEHR Composition version should always be created for each received version, even where the content does not change at all (e.g. a microbiology test where the result is “no growth” in both interim and final results).

**Duplicate Detection**

Another task is to disambiguate duplicates (often caused by failure of a network connection during sending) coming from the feeder system. In some cases however duplicates are erroneously given new ids by the feeder system, giving the receiver the impression of a new information. In such cases, a further item of meta-data may be required:

• hash or content signature generated (most likely by the converter) from the received information.

**Differentially Coded Data**

A further problem is that the originating system may send new versions of an item which are not complete in and of themselves, i.e. which only include new or changed elements with respect to a previous send of the same item. An example is a system which sends a correction to an HL7v2 blood test message, where the correction includes just the “serum sodium” data item. In this case, special processing will be required in the openEHR converter component, in order to regenerate a full data item from difference data when it is received. Such processing may also have to take account of deleted items.

In summary, the Feeder audit class design tries to accommodate the recording of as much of the above meta-data as is relevant in any particular case. It is up to the design of openEHR conversion front-end components as well as proper analysis of the situation to determine which identifiers are germane to the needs of traceability. In general, any meta-data of medico-legal significance should be captured where it is available.

**Using Feeder Audit in Converted Data**

Although the design of the openEHR converter is outside the scope of the current document, it is worth considering a common design approach, and where the FEEDER_AUDIT class fits in. An effective way of converting non-openEHR data such as HL7v2 messages, relational data etc, is in two steps. The first is to perform a ‘syntactic’ conversion to Compositions containing instances of the
The openEHR Common Information Model

**Generic Entry** class (described in the Integration IM), using ‘legacy archetypes’. The resulting database will contain versioned Compositions containing **generic_entry** instances; logically this database does not contain EHRs but simply external data converted to openEHR computational form. The relevant **feeder_audit** instances should be attached to the Compositions containing the corresponding **generic_entry** instances. The second step is to perform a ‘semantic’ conversion to subtypes of **entry**, i.e. **observation**, **evaluation**, **instruction** and **action**, according to standardised clinical archetypes. There are various possibilities for what to do with the Feeder audit. The minimum Feeder audit required on the final instance contains the originating system information, but none of the information to do with feeder or intermediate systems. This will satisfy medico-legal needs. Alternatively, a complete copy could be made, even though the feeder-related meta-data is probably only of use in the conversion environment. What the Feeder audit looks like in the EHR proper may depend on local legislation, norms or other factors. Completely alternative conversion processes are also possible, in which no intermediate form of data exists.

**Structural Correspondence**

There is no guarantee that the granularity of information recorded in the feeder system obeys the rules of Entries, Compositions, etc. As a consequence, feeder information might correspond to any level of information defined in the openEHR models. In order to be able to record feeder audit information correctly, the model has to be able to associate an audit trail with any granularity of object. For this reason, feeder audit information is attached to the **locatable** class via the **feeder_audit** attribute, even though it is preferable by design to have it attached to the equivalent of Compositions or at least the equivalent of archetype entities (i.e. Compositions, Section trees and Entries). Its usual usage is to attach it to the outermost object to which it applies. In other words, in most cases, during a legacy data conversion process, the entirety of a Composition needs only one **feeder_audit** to document its origins. In exceptional cases, where feeder data comes in in near real time, e.g. from an ICU database, separate **feeder_audit** objects may need to be generated for parts of a Composition; each commit in this situation will create a stack of versions of one Composition, with a growing number of **feeder_audit** objects attached to internal data nodes, each documenting the last import of data.

The Feeder audit information is included as part of the data of the Composition, rather than part of the audit trail of version committal, because it remains relevant throughout the versioning of a logical Composition, i.e. when a new version is created, the feeder information is retained as part of the current version to be seen and possibly modified, just as for the rest of its content. If the main part of the content is modified so drastically as to make the feeder audit irrelevant, it too can be removed.

A second consequence of feeder and legacy systems is that structural data items may need to be synthesised in order to create valid structures, even though the source system does not have them. For example, a system may have the equivalent data of Entries, but no Sections or other higher-level data items; these have to be synthesised during conversion. To indicate synthesis of a data node, a **feeder_audit** instance is attached to the **locatable** in question, and its **change_type** set to “synthesised”.

---

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
# Class Descriptions

## 3.2.1 Class LOCATABLE

<table>
<thead>
<tr>
<th>CLASS</th>
<th>LOCATABLE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Root structural class of all information models. The parent feature may be imple-</td>
</tr>
<tr>
<td></td>
<td>mented as a function or attribute.</td>
</tr>
<tr>
<td>GEHR</td>
<td>Name attribute in ARCHETYPED, meaning attribute in G1_PLAIN_TEXT.</td>
</tr>
<tr>
<td>Synapses</td>
<td>Each record component includes a Synapses Object ID attribute to reference the</td>
</tr>
<tr>
<td></td>
<td>Synapses Object (archetype) used as the basis for its construction. All record</td>
</tr>
<tr>
<td></td>
<td>components include a name attribute intended for the same purpose as the openEHR</td>
</tr>
<tr>
<td></td>
<td>equivalent.</td>
</tr>
</tbody>
</table>

### Abstract

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>parent: LOCATABLE</td>
</tr>
<tr>
<td></td>
<td>Parent of this node in compositional hierarchy.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1 uid: OBJECT_ID</td>
<td>Optional globally unique object identifier for root points of archetyped structures.</td>
</tr>
<tr>
<td>1..1 archetype_node_id: String</td>
<td>Design-time archetype id of this node taken from its generating archetype; used to build archetype paths. Always in the form of an “at” code, e.g. “at0005”. This value enables a &quot;standardised&quot; name for this node to be generated, by referring to the generating archetype local ontology. At an archetype root point, the value of this attribute is always the stringified form of the archetype_id found in the archetype_details object.</td>
</tr>
<tr>
<td>1..1 name: DV_TEXT</td>
<td>Runtime name of this fragment, used to build runtime paths. This is the term provided via a clinical application or batch process to name this EHR construct: its retention in the EHR faithfully preserves the original label by which this entry was known to end users.</td>
</tr>
<tr>
<td>0..1 archetype_details: ARCHETYPED</td>
<td>Details of archetyping used on this node.</td>
</tr>
</tbody>
</table>
CLASS | LOCATABLE (abstract)
--- | ---

| 0..1 | feeder_audit: FEEDER_AUDIT | Audit trail from non-openEHR system of original commit of information forming the content of this node, or from a conversion gateway which has synthesised this node. |
| 0..1 | links: Set <LINK> | Links to other archetyped structures (data whose root object inherits from ARCHETYPED, such as ENTRY, SECTION and so on). Links may be to structures in other compositions. |

FUNCTIONS | SIGNATURE | MEANING
--- | --- | ---
| 1..1 | is_archetype_root: Boolean | True if this node is the root of an archetyped structure. |
| | path_of_item (a_loc: LOCATABLE): String | The path to an item relative to the root of this archetyped structure. |
| | item_at_path (a_path: String): LOCATABLE | The item at a path (relative to this item). |
| | valid_path (a_path: String): Boolean | True if the path is valid with respect to the current item. |
| 0..1 | concept: DV_TEXT | Clinical concept of the archetype as a whole (= derived from the ‘archetype_node_id’ of the root node) |
| | require | is_archetype_root |

**Invariant**

\[ \text{Archetype\_node\_id\_valid}: \text{archetype\_node\_id} \neq \text{Void and then not} \]
\[ \text{archetype\_node\_id}.\text{is\_empty} \]
\[ \text{Name\_valid}: \text{name} \neq \text{Void} \]
\[ \text{Links\_valid}: \text{links} \neq \text{Void implies not} \text{links}.\text{empty} \]
\[ \text{Archetyped\_valid}: \text{is\_archetype\_root xor archetype\_details} = \text{Void} \]

### 3.2.2 ARCHETYPED Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Archetypes act as the configuration basis for the particular structures of instances defined by the reference model. To enable archetypes to be used to create valid data, key classes in the reference model act as “root” points for archetyping; accordingly, these classes have the archetype_details attribute set. An instance of the class ARCHETYPED contains the relevant archetype identification information, allowing generating archetypes to be matched up with data instances</td>
</tr>
<tr>
<td><strong>GEHR</strong></td>
<td>G1_ARCHETYPED</td>
</tr>
</tbody>
</table>
3.2.3 LINK Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPED</th>
</tr>
</thead>
</table>
| Synapses/SynEx | The SynEx approach does not distinguish between multiple layers of archetypes; hence an ‘archetype’ covers all information in an entire composition. Consequently, there is only one place where archetype identifiers in the openEHR sense are used (at the top); all other archetype identifiers are equivalent to the archetype_node_id attribute from LOCATABLE.

The Synapses ObjectID attribute provides a unique reference to each fine-grained element of an archetype, and is therefore also functionally equivalent to the archetype_id attribute at the root points in an openEHR structure. |
| CEN | The 1999 pre-standard does not include any equivalent to the archetype concept. However each architectural component must include a reference to an entry in the relevant normative table in the Domain Termlist pre-standard (part 2), to provide a high-level semantic classification of the component. All Architectural components include a component name structure to specify its label: the source of possible values for such a label was not clearly defined. The 2003 revision of ENV 13606 explicitly includes archetype identification attributes in the class RECORD_COMPONENT. |

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>archetype_id: ARCHETYPE_ID</td>
<td>Globally unique archetype identifier.</td>
</tr>
<tr>
<td>0..1</td>
<td>access_control: ACCESS_GROUP_REF</td>
<td>The access control settings of this component.</td>
</tr>
<tr>
<td>1..1</td>
<td>rm_version: String</td>
<td>Version of the openEHR reference model used to create this object. Expressed in terms of the release version string, e.g. “1.0”, “1.2.4”.</td>
</tr>
</tbody>
</table>

Invariant: archetype_id_valid: archetype_id /= Void
rm_version_valid: rm_version /= Void and then not rm_version.is_empty

3.2.3 LINK Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The LINK type defines a logical relationship between two items, such as two ENTRYs or an ENTRY and a COMPOSITION. Links can be used across compositions, and across EHRs. Links can potentially be used between interior (i.e. non archetype root) nodes, although this probably should be prevented in archetypes. Multiple LINKs can be attached to the root object of any archetyped structure to give the effect of a 1-&gt;N link</td>
</tr>
<tr>
<td>Use</td>
<td>1:1 and 1:N relationships between archetyped content elements (e.g. ENTRYs) can be expressed by using one, or more than one, respectively, DV LINKs. Chains of links can be used to see “problem threads” or other logical groupings of items.</td>
</tr>
<tr>
<td>CLASS</td>
<td>LINK</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>MisUse</td>
<td>Links should be between archetyped objects only, i.e. between objects representing complete domain concepts because relationships between sub-elements of whole concepts are not necessarily meaningful, and may be downright confusing. Sensible links only exist between whole ENTRYs, SECTIONs, COMPOSITIONs and so on.</td>
</tr>
<tr>
<td>CEN</td>
<td>The Link Item class is a simplified form of the Synapses Link Item, permitting links to be established but with limited labelling and no representation for importance.</td>
</tr>
<tr>
<td>Synapses</td>
<td>The Link Item class provides the means to link any arbitrary parts of a single EHR, for the overall linkage network to be labelled and revised, and for each direct link to be labelled explicitly. An importance attribute provides guidance on how links should be handled if only part of a linkage network is requested by a client process.</td>
</tr>
<tr>
<td>GEHR</td>
<td>n/a</td>
</tr>
<tr>
<td>HL7v3</td>
<td>The ACT_RELATIONSHIP class in some cases appears to correspond to LINK.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>meaning: DV_TEXT</td>
<td>Used to describe the relationship, usually in clinical terms, such as “in response to” (the relationship between test results and an order), “follow-up to” and so on. Such relationships can represent any clinically meaningful connection between pieces of information. Values for meaning include those described in Annex C, ENV 13606 pt 2 [11] under the categories of “generic”, “documenting and reporting”, “organisational”, “clinical”, “circumstancial”, and “view management”.</td>
<td></td>
</tr>
<tr>
<td>type: DV_TEXT</td>
<td>The type attribute is used to indicate a clinical or domain-level meaning for the kind of link, for example “problem” or “issue”. If type values are designed appropriately, they can be used by the requestor of EHR extracts to categorise links which must be followed and which can be broken when the extract is created.</td>
<td></td>
</tr>
<tr>
<td>target: DV_EHR_URI</td>
<td>The logical “to” object in the link relation, as per the linguistic sense of the meaning attribute.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariant</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Meaning_valid: meaning /= Void</td>
<td></td>
</tr>
<tr>
<td>Type_valid: type /= Void</td>
<td></td>
</tr>
<tr>
<td>Target_valid: target /= Void</td>
<td></td>
</tr>
</tbody>
</table>
### 3.2.4 FEEDER_AUDIT Class

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>originating_system_audit: FEEDER_AUDITDETAILS</code></td>
<td>Any audit information for the information item from the originating system.</td>
</tr>
<tr>
<td><code>originating_system_item_ids: List&lt;DV_IDENTIFIER&gt;</code></td>
<td>Identifiers used for the item in the originating system, e.g. filler and placer ids.</td>
</tr>
<tr>
<td><code>feeder_system_audit: FEEDER_AUDITDETAILS</code></td>
<td>Any audit information for the information item from the feeder system, if different from the originating system.</td>
</tr>
<tr>
<td><code>feeder_system_item_ids: List&lt;DV_IDENTIFIER&gt;</code></td>
<td>Identifiers used for the item in the feeder system, where the feeder system is distinct from the originating system.</td>
</tr>
</tbody>
</table>

**Invariants**

- `Originating_system_audit_valid:` `originating_system_audit /= Void`
### 3.2.5 FEEDER_AUDIT_DETAILS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>FEEDER_AUDIT_DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Audit details for any system in a feeder system chain. Audit details here means the general notion of who/where/when the information item to which the audit is attached was created. None of the attributes is defined as mandatory, however, in different scenarios, various combinations of attributes will usually be mandatory. This can be controlled by specifying feeder audit details in legacy archetypes.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>1</td>
<td>system_id: String</td>
</tr>
<tr>
<td>0..1</td>
<td>provider: PARTY_IDENTIFIED</td>
</tr>
<tr>
<td>0..1</td>
<td>location: PARTY_IDENTIFIED</td>
</tr>
<tr>
<td>0..1</td>
<td>time: DV_DATE_TIME</td>
</tr>
<tr>
<td>0..1</td>
<td>subject: PARTY_PROXY</td>
</tr>
<tr>
<td>0..1</td>
<td>version_id: String</td>
</tr>
<tr>
<td><strong>Invariants</strong></td>
<td>System_id_valid: system_id /= Void and then not system_id.is_empty</td>
</tr>
</tbody>
</table>
4  Generic Package

4.1  Overview
The classes presented in this section are abstractions of concepts which are generic to the domain of health (and most likely other domains), such as ‘participation’ and ‘attestation’. Here, “generic” means that the same model can be used, regardless of where they are contextually used in other models. The generic cluster is illustrated in FIGURE 4.

4.2  Design Principles
There are two ways to refer to an identity in the openEHR EHR: using PARTY_REF directly, which records an identifier of the party in some external system, and using PARTY_PROXY, which supports a small amount of descriptive data, depending on the subtype, and an optional PARTY_REF. The semantics of PARTY_REF are described in the Common IM, identification package, while the semantics of PARTY_PROXY and use of PARTY_REF in such entities are described below.
4.2.1 Referring to Demographic Entities

The approach taken in openEHR for representing demographic and user entities in the EHR data is based on the following assumptions:

- there is at least one human readable name or official identifier of the party, such as “Julius Marlowe, MD”, “NHS provider number 1039385”, or a system user id such as “Rahil.Azam”;
- there might be data in a service external to the EHR for the party in question, such as a demographic, identity management or patient index service; if there is, we want to reference it;
- the subject of the record is never to be identified in any direct way (i.e. via the use of her name or other human-readable details), but may include a meaningless identifier in some external system.

The PARTY_PROXY class and subtypes model references to parties based on these assumptions. The semantics of PARTY_PROXY enable a flexible approach: in stricter environments that have identity management and demographic services, and where there is an entry in such a service for the party in question, PARTY_PROXY:external_ref will be non-Void, while in other environments, it will be empty.

The two subtypes correspond to the mutually distinct categories of the ‘subject of the record’, known as the ‘self’ party in openEHR, and any other party. Whenever the record subject has to be referred to in the record, an instance of PARTY_SELF is used, while PARTY_IDENTIFIED is used for all other situations. The latter class provides for optional human-readable names and formal identifiers, each keyed by purpose or meaning.

The RELATED_PARTY type is used whenever the relationship of the party to the record subject is required. Relationships are coded and include familial ones (‘mother’, ‘uncle’, etc) as well as relationships like ‘donor’, ‘travelling companion’ and so on.

PARTY_SELF and Referring to the Patient from the EHR

There are three schemes which are likely to be used for referring to patient (i.e. the record subject) demographic or patient master index (PMI) data from within the EHR, each likely to be valid in different circumstances. These are as follows.

- Once only in EHR.subject using the PARTY_SELF:external_ref. Since the EHR object is separate from the EHR contents, the root instance of PARTY_SELF will generally not be visible.
- Setting the external_ref in every instance of PARTY_SELF; this solution makes the patient external ref visible in every instance of PARTY_SELF, which is reasonable in a secure environment, and convenient for copying parts of the record around locally.
- On no instances of PARTY_SELF. This is the most secure approach, and means that the link between the EHR and the patient has to be done outside the EHR, by associating EHR.uid and the patient demographic/PMI identifier. This approach is more likely for more open environments.

All three schemes are supported by the openEHR model, and will probably all find use in different settings and EHR deployment types.

4.2.2 Participation

The Participation abstraction models the interaction of some Party in an activity. In the openEHR reference models, participations are actually modelled in two ways. In situations where the kinds of participation are known and constant, they are modelled as a named attribute in the relevant reference
model. For example, the committer: PARTY_PROXY attribute in AUDIT_DETAILS models a participation in which the function is “committal”. Where the kind of participation is not known at design time, a generic PARTICIPATION class is used. This class refers to a Party via a PARTY_PROXY instance, and records the function, time interval and (coded) mode of the participation. It can be used in any other reference model as required.

4.2.3 Audit Information

Audit Details
Three classes are provided to represent audit information. The first, AUDIT_DETAILS expresses the details that would be captured about a user when committing some information to a repository of some kind, which may be version controlled. It records committer, time, change type and description. Committer is recorded using a PARTY_PROXY, allowing for PARTY_SELF to be used when the committer is the record subject, and for other identifying information to be included for other users, expressed using PARTY_IDENTIFIED. The kind of identifying information used in PARTY_PROXY instances in AUDIT_DETAILS may be different from that used in COMPOSITION.composer or elsewhere, i.e. in the form of a system login identifier, e.g. “maxime.lavache@stpatricks.health.ie”.

Revision History
The classes REVISION_HISTORY and REVISION_HISTORY_ITEM express the notion of a revision history, which consists of audit items, each associated with a revision number. An instance of the REVISION_HISTORY_ITEM class is designed to express the information that corresponds to an item in a revision history, i.e. a list of all audits relating to some information item. The version_id is included to indicate which revision each audit corresponds to. These classes provide an interoperable definition of revision history for the VERSIONED_OBJECT and AUTHORED_RESOURCE classes.

4.2.4 Attestation
Attestation is another concept which occurs commonly in health information. An attestation is an explicit signing by one healthcare agent of particular content for various particular purposes, including:

- for authorisation of a controlled substance or procedure (e.g. sectioning of patient under mental health act);
- witnessing of content by senior clinical professional;
- indicating acknowledgement of content by intended recipient, e.g. GP who ordered a test result.

Here it is modelled as a subtype of AUDITDETAILS, meaning that it is logically a kind of audit, with additional information pertinent to the act of signing, namely a proof object and the list of identifiers of the attested items. At a minimum, the proof should be a digital certificate which binds the following items together:

- the identity of the attesting party;
- the thing attested to, e.g. a statement like “Do you agree that the form below is an accurate record of the clinical session just completed?”, and potentially a hash or other compressed, encoded representation of the attested-to content;
- the time;
- appropriate digital signatures.

Such a certificate may be included in the record, or it may exist in some other place such as a notary service or similar. The use of the DV_ENCAPSULATED type for the proof attribute allows for either.
Normally the list of items should be a single Entry or Composition, but there is nothing stopping it including fine-grained items, even though separate attestation of such items does not appear to be commensurate with good clinical information design or process.

The reason attribute is used to indicate why the attestation occurred, and is coded using the openEHR Terminology group “attestation reason”, which includes values such as “authorisation” and “witnessed”. The is_pending attribute marks the attestation as either having been done or awaiting completion depending on its value. This facilitates querying the record to find items needing to be signed or witnessed. When an attestation is required, the most common scenario will be that a Composition Version will be committed with an audit of type ATTESTATION, rather than just AUDIT_DETAILS; the is_pending flag will be set to True to indicate that the committed information needs to be signed by another person. When the latter happens, it will cause a new ATTESTATION object to be added to the VERSION. attestations list, this time with is_pending set to False, and the appropriate proof supplied. In general, a need for attestation followed by an attestation will be represented by two ATTESTATION objects, the first in the is_pending state, and the second with is_pending False, and containing the relevant proof.

4.3 Class Descriptions

4.3.1 PARTY_PROXY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTY_PROXY (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract concept of a proxy description of a party, including an optional link to data for this party in a demographic or other identity management system. Sub-typed into PARTY_IDENTIFIED and PARTY_SELF.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>0..1</td>
<td>external_ref: PARTY_REF</td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 PARTY_SELF Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTY_SELF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Party proxy representing the subject of the record.</td>
</tr>
<tr>
<td>Use</td>
<td>Used to indicate that the party is the owner of the record. May or may not have external_ref set.</td>
</tr>
<tr>
<td>Inherit</td>
<td>PARTY_PROXY</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 4.3.3 PARTY_IDENTIFIED Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTY_IDENTIFIED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Proxy data for an identified party other than the subject of the record, minimally consisting of human-readable identifier(s), such as name, formal (and possibly computable) identifiers such as NHS number, and an optional link to external data. There must be at least one of <code>name</code>, <code>identifier</code> or <code>external_ref</code> present.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Used to describe parties where only identifiers may be known, and there is no entry at all in the demographic system (or even no demographic system). Typically for health care providers, e.g. name and provider number of an institution.</td>
</tr>
<tr>
<td><strong>Misuse</strong></td>
<td>Should not be used to include patient identifying information.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>PARTY_PROXY</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1 (cond) name: String</td>
<td>Optional human-readable name (in String form).</td>
</tr>
<tr>
<td>0..1 (cond) identifiers: List&lt;DV_IDENTIFIER&gt;</td>
<td>One or more formal identifiers (possibly computable).</td>
</tr>
</tbody>
</table>

**Invariant**

- `Basic_valid` name /= Void or identifiers /= Void or external_ref /= Void
- `Name_valid`: name /= Void implies not name.is_empty
- `Identifiers_valid`: identifiers /= Void implies not identifiers.is_empty

### 4.3.4 PARTY_RELATED Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTY_RELATED</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Denote a party and its relationship to the subject of the record.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Use where the relationship between the party and the subject of the record must be known.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>PARTY_IDENTIFIED</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 relationship: DV_CODED_TEXT</td>
<td>Relationship of subject of this ENTRY to the subject of the record. May be coded. If it is the patient, coded as “self”.</td>
</tr>
</tbody>
</table>
4.3.5 PARTICIPATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>PARTICIPATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model of a participation of a Party (any Actor or Role) in an activity.</td>
</tr>
<tr>
<td>Use</td>
<td>Used to represent any participation of a Party in some activity, which is not explicitly in the model, e.g. assisting nurse. Can be used to record past or future participations.</td>
</tr>
<tr>
<td>Misuse</td>
<td>Should not be used in place of more permanent relationships between demographic entities.</td>
</tr>
<tr>
<td>HL7v3</td>
<td>RIM Participation class.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>performer: PARTY_PROXY</td>
<td>The id and possibly demographic system link of the party participating in the activity.</td>
</tr>
<tr>
<td></td>
<td>function: DV_TEXT</td>
<td>The function of the Party in this participation (note that a given party might participate in more than one way in a particular activity). This attribute should be coded, but cannot be limited to the HL7v3:ParticipationFunction vocabulary, since it is too limited and hospital-oriented.</td>
</tr>
<tr>
<td>1..1</td>
<td>mode: DV_CODED_TEXT</td>
<td>The mode of the performer / activity interaction, e.g. present, by telephone, by email etc.</td>
</tr>
<tr>
<td>0..1</td>
<td>time: DV_INTERVAL &lt;DV_DATE_TIME&gt;</td>
<td>The time interval during which the participation took place, if it is used in an observational context (i.e. recording facts about the past); or the intended time interval of the participation when used in future contexts, such as EHR Instructions.</td>
</tr>
</tbody>
</table>

| Invariant | Performer_valid: performer /= Void |
|           | Function_valid: function /= Void and then function.generating_type.is_equal(“DV_CODED_TEXT”) implies terminology(“openehr”).codes_for_group_name(“participation function”, “en”) .has(function.defining_code) |
|           | Mode_valid: mode /= Void and terminology(“openehr”).codes_for_group_name(“participation mode”, “en”).has(mode.defining_code) |
4.3.6 AUDITDETAILS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>AUDITDETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The set of attributes required to document the committal of an information item to a repository.</td>
</tr>
<tr>
<td>Synapses</td>
<td>Composition class</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_COMMIT_AUDIT</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>system_id: String</td>
<td>Identity of the system where the change was committed. Ideally this is a machine- and human-processable identifier, but it may not be.</td>
</tr>
<tr>
<td>committer: PARTY_PROXY</td>
<td>Identity and optional reference into identity management service, of user who committed the item.</td>
</tr>
<tr>
<td>time_committed: DV_DATE_TIME</td>
<td>Time of committal of the item.</td>
</tr>
<tr>
<td>change_type: DV_CODED_TEXT</td>
<td>Type of change. Coded using the openEHR Terminology “audit change type” group.</td>
</tr>
<tr>
<td>description: DV_TEXT</td>
<td>Reason for committal.</td>
</tr>
</tbody>
</table>

**Invariants**

| System_id_valid: system_id /= Void and then not system_id.is_empty |
| Committer_valid: committer /= Void |
| Time_committed_valid: time_committed /= Void |
| Change_type_valid: change_type /= Void and then terminology(“openehr”).codes_for_group_name(“audit change type”, “en”).has(change_type.defining_code) |

4.3.7 ATTESTATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ATTESTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Record an attestation of a party (the committer) to item(s) of record content. The type of attestation is</td>
</tr>
<tr>
<td>Inherit</td>
<td>AUDITDETAILS</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>proof: DV_ENCAPSULATED</td>
<td>Proof of attestation.</td>
</tr>
</tbody>
</table>
4.3.8 REVISION_HISTORY Class

Purpose

Defines the notion of a revision history of audit items, each associated with the version for which that audit was committed.

Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>items: List</td>
<td>The items in this history.</td>
</tr>
</tbody>
</table>

Invariants

| Items_valid: items /= Void |
### 4.3.9 REVISION_HISTORY_ITEM Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>REVISION_HISTORY_ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>An entry in a revision history, corresponding to a version from a versioned container. Consists of <code>AUDITDETAILS</code> instances with revision identifier of the revision to which the <code>AUDITDETAILS</code> intance belongs.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>audits: 1..1</td>
<td>Set&lt;AUDITDETAILS&gt;</td>
</tr>
<tr>
<td>version_id: 1..1</td>
<td>OBJECT_VERSION_ID</td>
</tr>
<tr>
<td><strong>Invariants</strong></td>
<td>Audit_valid: audit /= Void and then not audits.is_empty</td>
</tr>
<tr>
<td></td>
<td>Version_id_valid: version_id /= Void</td>
</tr>
</tbody>
</table>
5 Directory Package

5.1 Overview

The directory package is illustrated in FIGURE 5. It provides a simple abstraction of a versioned folder structure. The VERSIONED_FOLDER class is the binding of VERSIONED_OBJECT<T> to FOLDER, i.e. it is a VERSIONED_OBJECT<FOLDER>. This means that each of its versions is a FOLDER structure. It provides a means of versioning FOLDER structures over time, which is useful in the EHR, Demographics service or anywhere else where Folders are used to group things. A FOLDER instance is simple: it contains more FOLDERS and/or items, which are references to other (usually versioned) objects. A FOLDER structure is therefore like a directory containing references to objects. Since they are only references, multiple references to the same object are possible, allowing the structure to be used to multiply classify other objects. If it is used with VERSIONED_COMPOSITIONs for example, the folders might be used to represent episodes and at the same time problem groups.

FOLDER structures inside the VERSIONED_FOLDER are archetypable structures, and FOLDER archetypes can be created in the same fashion as say SECTION archetypes for the EHR.

5.1.1 Paths

Directory paths are built using the name attribute values inherited from LOCATABLE into each FOLDER object. In real data, these will usually be derived from the value of the archetype_node_id attribute, plus a uniqueness modifier if required. Example paths:

    /folders[hospital episodes]/items[]
    /folders[patient entered data]/folders[diabetes monitoring]
    /folders[homeopathy contacts]

Uniqueness modifiers are appended in brackets, and are only needed to differentiate folders at the same node that would otherwise have the same names, e.g.

    [hospital episodes]
    [hospital episodes(car accident Aug 1998)]
5.2 Class Descriptions

5.2.1 VERSIONED_FOLDER Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSIONED_FOLDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A version-controlled hierarchy of FOLDERs giving the effect of a directory.</td>
</tr>
<tr>
<td>Inherit</td>
<td>VERSIONED_OBJECT &lt;FOLDER&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

5.2.2 FOLDER Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>FOLDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The concept of a named folder.</td>
</tr>
<tr>
<td>CEN</td>
<td>FOLDER class</td>
</tr>
<tr>
<td>Synapses</td>
<td>RecordFolder class</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>folders: List&lt;FOLDER&gt;</td>
<td>Sub-folders of this FOLDER.</td>
</tr>
<tr>
<td>0..1</td>
<td>items: List&lt;OBJECT_REF&gt;</td>
<td>The list of references to other (usually) versioned objects logically in this folder.</td>
</tr>
</tbody>
</table>

| Invariants | Folders_valid: folders /= Void implies not folders.empty |


6 Change Control Package

6.1 Overview

In various openEHR reference models, the semantics of formal change control are required. There are two architectural aspects of managing changes to data. The first is the concept of a complex information object, being versioned in time, meaning that its creation and all subsequent modifications cause new “versions” to be created, rather than literally overwriting the existing data. Each version includes an audit trail, typically containing the identity of a user, the date/time of the change, and a reason for the change. The second aspect recognises that repositories are made up of complex information objects, and that changes are not in fact just made to individual objects, but to the repository itself. Any change by a user may change more than one versioned object in the repository, and the set of such changes - a “change-set” - constitutes the logical unit of change to the repository, taking it from one valid state to the next.

These concepts are well-known in configuration management (CM), and are used as the basis for most software and other change management systems, including numerous free and commercial products available today.

The following sections describe the configuration management paradigm in more detail, and explain how it relates to the openEHR reference models, in particular, the model for the EHR.

6.2 Design Principles

6.2.1 The Configuration Management Paradigm

The “configuration management” (CM) paradigm is well-known in software engineering, and has its own IEEE standards. CM is about managed control of changes to a repository of items (formally called “configuration items” or CIs), and is relevant to any logical repository of distinct information items which changes in time. In health information systems, at least two types of information require such management: electronic health records, and demographic information. In most analyses in the past, the need for change management has been expressed in terms of specific requirements for audit trailing of changes, availability of previous states of the repository and so on. Here, we aim to provide a formal, general-purpose model for change control, and show how it applies to health information.

6.2.2 Organisation of the Repository

The general organisation of a repository of complex information items such as a software repository, or the EHR consists of the following:

- a number of distinct information items, or configuration items, each of which is uniquely identified, and may have any amount of internal complexity;
- optionally, a directory system of some kind, in which the configurations items are organised.

Thus, in a software or document repository, the CIs are files arranged in the directories of the file system; in an EHR based on the GEHR or CEN models, they are Compositions arranged in a Folder structure. Contributions are made to the repository by users. This general abstraction is visualised in FIGURE 6.

6.2.3 Change Management

As implied earlier, change doesn’t occur to CIs in isolation, but to the repository as a whole. Possible types of change include:
6.2.4 Changes in Time

Properly managing changes to the repository requires two mechanisms. The first, *version control*, is used to manage versions of each CI, and of the directory structure if there is one. The second is the concept of the “change-set”, or what we will call a *contribution*. This is the set of changes to individual CIs (and the directory structure) made by a user as part of some logical change. For example, in a document repository, the logical change might be an update to a document that consists of multiple files (CIs). There is one contribution, consisting of changes to the document file CIs, to the repository. In the EHR, a contribution might consist of changes to more than one Composition, and possibly to the organising Folder structure.

A typical sequence of changes to a repository is illustrated below. FIGURE 7 shows the effect of four Contributions (indicated by blue ovals on the left hand side) to a repository containing a number of CIs (that the directory tree is not shown for the sake of simplicity). As each Contribution is made, the repository is changed in some way. The first brings into existing a new CI, and modifies three others (changes indicated by the ‘C’ triangles). The second contribution causes the creation of a new CI.
FIGURE 7 Contributions to the Repository (delta form)

only. The third causes a creation as well as two changes, while the fourth causes only a change. (Again, changes to the folder structure are not shown here).

One nuance which should be pointed out is that, in FIGURE 7, contributions are shown as if they are literally a set of deltas, i.e. exactly the changes which occur to the record. Thus, the first contribution is the set \(\{CI_w, Ca_1, Cc_1, Cd_1\} \) and so on. Whether this is exactly true depends on the construction of applications. In some situations, some CIs may be updated by the user viewing the current list and entering just the changes - the situation shown in FIGURE 7; in others, the system may provide the current state of these CIs for editing by the user, and submit the updated versions, as shown in FIGURE 8. Some applications may do both, depending on which CI is being updated. The internal versioning implementation may or may not generate deltas as a way of efficient storage.

For our purposes here, we consider a contribution as being the logical set of CIs changed or created at one time, as implied by FIGURE 8.

### 6.2.5 General Model of a Change-controlled Repository

FIGURE 9 shows an abstract model of a change-controlled repository, which consists of:

- version-controlled configuration items - instances of `VERSIONED_OBJECT<T>`;
- `CONTRIBUTION`s;
- an optional directory system of folders. If folders are used, the folder structure must also be versioned as a unit.

The actual type of links between the controlled repository and the other entities might vary - in some cases it might be composition, in others aggregation; cardinalities might also vary. FIGURE 9 there-
fore provides a guide to the definition of actual controlled repositories, such as an EHR, rather than a formal specification for them.

6.3 Formal Model

6.3.1 Overview

FIGURE 10 illustrates a formal model of a version repository. In this model, the class VERSIONED_OBJECT<T> provides the versioning facilities for one CI, such as an EHR Composition, or a Party in a demographic system. Each version is an instance of the class VERSION<T> or its sub-type IMPORTED_VERSION<T>, which combines the data being versioned, audit trails, and any attestations applied to the version. The classes VERSIONED_OBJECT<T>, VERSION<T>, MERGED_VERSION<T> and IMPORTED_VERSION<T> are generic classes, with the generic parameter type T being the type of the data. This ensures that all versions in a given VERSIONED_OBJECT are of the same type, such as COMPOSITION, FOLDER, or PARTY and that the version container itself is properly typed.
Each VERSIONED_OBJECT has a unique identifier recorded in the uid attribute (a HIER_OBJECT_ID typically containing a GUID), and a reference to the owning object (e.g. the owning EHR) in the owner_id attribute (this is typically also a GUID). The latter helps ensure that in storage systems, versioned objects are always correctly allocated to their enclosing repository, such as an EHR.

The data in a VERSIONED_OBJECT is a collection of instances of VERSION<T> and its subtypes, and is specified only in terms of its functional interface. How the representation of this collection is implemented inside the VERSIONED_OBJECT is not defined by this specification, only the form of any given Version is. Implementations of VERSIONED_OBJECT might range from the simple (all versions stored as full copies in a List or Set) to a sophisticated compressed versioning approach as used in software file version control and some object databases. (Note that an interoperable expression of VERSIONED_OBJECT is provided in the EHR Extract IM.)

The class VERSION<T> defines the data and meta-data of a single version in a VERSIONED_OBJECT. The uid attribute, of type OBJECT_VERSION_ID, contains the attributes owner_id, creating_system_id, and version_tree_id, and constitutes a globally unique version identifier enabling Versions to be identified and merged correctly regardless of copying and subsequent changes. The details of the version identification scheme are described below. The attribute preceding_version_id indicates which Version elsewhere in the tree is considered the previous one to the current one; in other words, which Version was “checked-out” to create the current Version.

### 6.3.2 Audits

Audits are recorded in the form of instances of the class AUDIT_DETAILS (common.generic package), which defines a set of attributes which form an audit trail, namely creating_system_id, committer, time_committed, change_type, and description. When a new Version is created locally, the
The \textit{commit\_audit} attribute is set to contain an audit trail corresponding to the local act of committal. All \texttt{VERSIONs} in all \texttt{VERSIONED\_OBJECTs} in any \textit{openEHR} deployment obey this rule.

However, if the Version being committed is a 1st-generation copy of a \texttt{VERSION} originally created and commited elsewhere, it is committed locally as an instance of the \texttt{IMPORTED\_VERSION} class, with the \texttt{original\_create\_audit} attribute of this class being set to be the \texttt{commit\_audit} from the received \texttt{VERSION} (i.e. the audit corresponding to the original creation of the item), and the \texttt{commit\_audit} value being set in the usual way. If on the other hand the received Version is actually an \texttt{IMPORTED\_VERSION} (meaning it has already been copied elsewhere), the \texttt{original\_create\_audit} attribute remains intact, and the \texttt{commit\_audit} attribute is set to record the local committal.

The function \texttt{create\_audit} is defined on the class \texttt{VERSION} to return the audit corresponding to the creation of the item; this will be the \texttt{commit\_audit} in the case of locally created \texttt{VERSIONs}, and \texttt{original\_create\_audit} in the case of \texttt{IMPORTED\_VERSIONs}.

This scheme ensures that two audits are guaranteed to be available for any Version: the audit corresponding to the action of committal to the local system, and the audit corresponding to the initial creation and committal of the object, regardless of how many times the items is copied. It also guarantees that locally created and imported information can be easily distinguished.

The \texttt{CONTRIBUTION} class also contains an \texttt{audit} attribute. Whenever a \texttt{CONTRIBUTION} is committed, this attribute captures to the time, place and committer of the committal act; these three attributes (\texttt{creating\_system\_id}, \texttt{committer}, \texttt{time\_committed} of \texttt{AUDIT\_DETAILs}) are copied into the corresponding attributes of the \texttt{commit\_audit} of each \texttt{VERSION} included in the \texttt{CONTRIBUTION}. This is done to enable sharing of versioned entities independently of which Contributions they were part of.

### 6.3.3 Version Lifecycle

Versioned content has a lifecycle state associated with it, modelled using the \texttt{VERSION.lifecycle\_state} attribute, which is coded from the \textit{openEHR} Terminology “version lifecycle state” group. The possible values include “incomplete”, “awaiting\_approval”, “complete” and “deleted”, each corresponding to a specific state of the content. Generally content will be committed in the “finished” state. However, in some circumstances, e.g. because the author has run out of time to finish writing part of the Composition, or due to an emergency, it may be committed as “unfinished” meaning that it is incomplete, or at least, unreviewed. In hospitals this is a common occurrence. Unfinished Compositions cannot be saved locally on the client machine, since this represents a security risk (a small client-side database would be much easier to hack into than a secure server). They must therefore be persisted on the server, either in the actual EHR, or in a 'holding bay' which was recognised as not being part of the EHR proper. Either way, the author would have to explicitly retrieve the Composition(s) and after further work or review, 'promote' them into the EHR as 'active' Compositions; alternatively, they might decide to throw them away.

Compositions which need approval (e.g. to be seen and signed by a more senior staff member) are saved in the “awaiting\_attestation” state.

Going from “unfinished” to “finished” almost always corresponds to a change in content, and corresponds to a new \texttt{VERSION} regardless. This modelling approach allows such content to exist on the EHR system, but to be flagged as unfinished when viewed by a user.

The \texttt{lifecycle\_state} values are given by the “version lifecycle state” group in the \textit{openEHR} Terminology.
6.3.4 Attestation

The *attestations* attribute allows attestations to be associated with the data in the version. Attestations are treated in *open*EHR as a kind of audit, with additional attributes, and are described in detail in the Common IM. They can be used as required by enterprise processes or legislation, and indicate who and when the item in question was attested. A digital “proof” is also required, although no assumption is made about the form of such proof. Normally, attestations refer to the entire version to which they are attached. However, it is possible for an *attestation* instance to refer to some finer-grained item within the data of the version, such as a single *ENTRY* within a *COMPOSITION*. If in subsequent versions, such an item is not changed (e.g. a different *ENTRY* in the same *COMPOSITION* is altered), then the relevant *VERSION* instances also need to refer to the *attestation* instances which remain valid. Since *attestations* are considered immutable objects once created, it does not really matter whether this is done by referring to a shared *attestation* instance, or by the use of copies.

Scenarios relating to attestation may cause attestations to be created at different times with respect to the committal of data to the EHR, as follows:

- **at committal**: highly sensitive information is to be added to the EHR, e.g. recording the fact of sectioning of a patient under the mental health act, diagnosis of a fatal disease etc. In this case, attestation is added at committal to the EHR;
- **post-committal**: a data-entry person e.g. a secretary, transcriptionist or student is responsible for entering the data, including routine things such as referrals, discharge summaries etc., which need to be verified by the relevant clinician; this may occur after committal to the EHR in some cases, leading to the temporary presence of entries "awaiting attestation" in the record.

As a result of these requirements, the model allows any number of attestations (from 0 to many) to be associated with each version of a versioned object. Attestations are considered to be neither part of the content, nor part of the audit information, but an external artifact which refers in to versions of versioned items. Attestations can be added at any time.

The class *CONTRIBUTION* defines the common audit information for the set of versions added to the repository due to a given contribution as well as a *description* of the contribution as a whole. *Contributions* refer to their member *VERSION* objects via *OBJECT_IDs*; similarly, the audit object of any *VERSIONABLE* refers to its creating *CONTRIBUTION* using an *OBJECT_IDs* reference.

These classes can be used to provide versioning and contributions in repositories such as an EHR, or a demographic repository. In the EHR reference model for example, to obtain a versioned Composition, the type *VERSIONED_OBJECT<COMPOSITION>* is defined.

6.3.5 Semantics of Copying in Distributed Systems

In *open*EHR, the only unit of copying that satisfies traceability requirements and can be used among distributed *open*EHR servers is the *VERSION* (or *IMPORTED_VERSION*). In order to copy a *COMPOSITION* or even an *OBSERVATION* somewhere else and retain versioning capability, its enclosing *VERSION<T>* object must be sent. In the case of the *COMPOSITION* type, a *VERSION<COMPOSITION>* (or *IMPORTED_VERSION<COMPOSITION>* ) object is sent. At the receiving system two things can happen.

If it is the first time any version of the item logically identified by its *VERSIONED_OBJECT.uid* was received from the originating system, a new *VERSIONED_OBJECT<T>* (e.g. *VERSIONED_OBJECT<COMPOSITION>* ) is created, with its *uid* set to the same value as the received *VERSION.owner_id* (i.e. the same as the relevant *VERSIONED_OBJECT.uid* in the originating system). This establishes the newly created *VERSIONED_OBJECT* as being the same as the one from which it
was copied. An IMPORTED_VERSION instance is then generated from the received instance, and committed in the normal way (i.e. as part of a Contribution), with its \texttt{uid.version_tree_id} and \texttt{uid.creating_system_id} attributes carrying their original values, and the audits being set as described above. Thus, if a VERSION was received with \texttt{uid.version_tree_id} = “2” and \texttt{uid.creating_system_id} = “au.gov.health.rdh.ehr3”, these details will be retained in the target system (whose system id might be “au.gov.health.svh.svrB”).

The alternative situation is if some version of the original item had been copied earlier. In this case, the newly received VERSION (or IMPORTED_VERSION) will be committed as an IMPORTED_VERSION to the VERSIONED_OBJECT that must already have been created, with its \texttt{uid.version_tree_id} and \texttt{uid.creating_system_id} attributes intact.

In most cases, the received information will remain as is for the duration. However, in some cases, users at the receiver system might want to make modifications as well. This is likely to happen in the case of information items representing things like medication lists and allergies. When new versions are added locally to a copied object, branching numbering is used in the \texttt{uid.version_tree_id}, while the local system id is recorded in the \texttt{uid.creating_system_id} attribute. This version identification scheme is described below.

### 6.3.6 Distributed Version Identification

The following version identification scheme is adapted from the work of Hnítynka and Plášil [3].

**Local Versioning**

The \texttt{version_tree_id} attribute of the \texttt{uid} of a VERSION identifies a version of an item with respect to other versions in the same tree. The requirements here are the same as for typical versioning systems in use in software configuration management, and are as follows:

- to encode the relationship between versions in the version id, that is to say, version ids are constructed that given a series of ids, the relative positions in the tree can be determined;
- to allow for branches, so that variants of a particular node can be created; this might be done due to translation, or for training purposes.

A suitable scheme satisfying the above requirements for health information is the simplest possible, i.e. a single number representing the version. Version identifiers thus start at 1 and continue by single increments. The succession of version identifiers formed by changes over time is known as the “trunk” of the version tree.

To support branching, a further pair of numbers is added. The first number identifies the branch (e.g. the 1st branch, 2nd branch etc from that trunk node), while the second identifies the version; this latter starts at 1, in the same way as the trunk version number. Both of these numbers also start at 1. The result of this is that version numbers like 1.1.1 (first version of first branch from trunk node 1), 2.3.3 (3rd version of 3rd branch from trunk node 2) become possible. Inside openEHR systems where sharing with other systems does not occur, it is expected that branched versioning will be used extremely rarely; translation is likely to be the only reason (for example if a portuguese translation of an english language version is made).

**Distributed Versioning**

However, in a distributed environment where copying and subsequent modification can be made, there are more requirements of a versioning scheme, as follows:

- it must be possible for an item to be copied and for local modifications then to be made;
• it must be possible to send more recent versions from the original system to a target system that has already received earlier versions, and for these versions to be distinguishable from versions in the existing system, including the previously imported versions - this enables the receiving system to know how and where to commit the received version;
• it must be guaranteed that any version of any object is uniquely identified globally, no matter whether it is a locally created trunk version, a locally created branch version or a change made to a copied version.

To satisfy these needs, two modifications are made to the identification scheme. The first is in the form of the creating_system_id attribute of the uid object, which represents the system where the version was created. This is a machine processable identifier, such as a reverse internet address or GUID. Whenever a new version of a particular Versioned object (with a particular uid) is created locally, creating_system_id is set to the identifier of the local system; if the version was imported, creating_system_id will be set to the identifier of the system of original creation. The second is to require branching version identifiers to be used when local modifications are made to copied versions from elsewhere; this ensures that the modifications now being made in the target system are considered in a global sense as logical branches or variants rather than trunk versions which are made in the originating system. It also allows later trunk versions from the originating system to be copied at some future time to the target system without version number clashes. With these changes, versions can be globally uniquely identified with the tuple \{owner_id, creating_system_id, version_tree_id\}.

The effect of the versioning scheme in practice is illustrated in FIGURE 11. On the left hand-side of the figure, a version container (i.e. an instance of VERSIONED_OBJECT) with uid=1 is shown; the first Version has uid.creating_system_id=“sysA”; uid.version_tree_id=“1”. Further local trunk and branch versions are also shown.

When the first Version is copied (copy #1) to system B, it is committed as an IMPORTED_VERSION to a VERSIONED_OBJECT which is effectively a copy of the original, and retains the same

---

**FIGURE 11** Versioning in a distributed environment
uid.version_tree_id and uid.creating_system_id. Subsequent copies (copy #2 and copy #3) can be made of later versions from system A to system B, with the effect that the version tree can be recreated inside system B (if required; there is of course no obligation to do anything with the received information). Users in system B can also make modifications to the received Version copies; these modifications are shown in grey, as branched versions with uid.creating_system_id = sysB. Independently, users in system B will of course be creating other content locally, e.g. as shown on the right-hand side, where a Versioned object with uid=2 has been created. Two places are indicated on the diagram where identification clashes could have occurred, but are prevented due to the use of the 3-part unique Version identifier tuple.

Two rules are required to make this system work, as follows:

- when copies of versions are made to another system that are intended to be committed by that system, branch versions from the originating system cannot be copied without their corresponding trunk versions;
- no system should create a new Versioned object (with a new uid) without first determining that it does not already have one with the same uid. This will should happen automatically if GUIDs are being used (and the generating software is reliable); checks may have to be made if ISO Oids are being used.

6.3.7 Semantics of Merging Versions

One of the most common operations in distributed versioned environments, particularly in healthcare, is that content created in one system is imported into another system, modifications are created locally there which are then sent back the first system. This information pathway corresponds to scenarios such as the patient being referred from primary care into a hospital, and later being discharged into primary (or other care). The usual need when the first system receives changes made to the data by the second system is to merge them back into the trunk of the version tree. Logically a ‘merge’ is the operation of using two versions of the same content to create a third version. How the source versions are used will vary based on the semantics of the information; it could be that the either is simply taken in its totality and the other discarded, or some mixture might be created of the two, in a process of editing by the user. In many cases in health, where the content is a medication or problem list, the user in the original system will review the received content and create a new trunk version locally using that content, since it will be deemed to be the most accurate of in the clinical computing environment. This scenario is illustrated in FIGURE 12.

In this figure, versions 1 and 2 of the content (e.g. a medication list) from Versioned object with uid=1 are copied from system A (e.g. a GP) to system B (e.g. a hospital). In system B, changes are made to version 2, creating a branch (as an instance of IMPORTED_VERSION<T>) as required by the copying rules described above. These changes (modified medication list) are then imported back into system A. The system A user performs a merge operation to create a new trunk version 3, using the sysB/2.1.2 and sysA/2 content; most likely, he/she simply reviews the two input versions and uses the sysB/2.1.2 content unchanged (the result is that system A now has an up-to-date medication list for the patient, including medications originally recorded at system A, as well as additions recorded at system B). The new Version is an instance of MERGED_VERSION<T>, with its input_ids attribute set to include the OBJECT_VERSION_ID representing sysB/2.1.2 (it does not need to include sysA/2, since this is already known in the preceding_version).

6.3.8 Semantics of Moving Version Containers

It will not be uncommon that whole VERSIONED_OBJECTS need to be moved to another system, e.g. due to a move of a complete patient record (due to the patient moving), or even the relocation of an
The semantics of a move are different from those of copying: with a move, there is no longer a source instance, and the destination instance becomes the primary instance. When the move is effected, the identifier of the system in which the VERSIONED_OBJECT now exists will usually be different from what it was before. As a consequence, subsequent versions of the content created in a moved version container will now have the uid.creating_system_id set to the id of new system only (rather than the concatenation of the new and old system ids, as happens in the copying case). This creates another variation on the version lineage, one in which the uid.creating_system_id value can change in the trunk line, as shown in FIGURE 13.

6.3.9 Transaction Semantics of Contributions

In terms of database management, Contributions are similar to nested transactions. An attempt to commit a Contribution should only succeed if each Version in the Contribution is committed successfully.
6.4 Class Descriptions

6.4.1 VERSIONED_OBJECT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSIONED_OBJECT&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Version control abstraction, defining semantics for versioning one complex object.</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td>1..1</td>
<td>uid: HIER_OBJECT_ID</td>
</tr>
<tr>
<td>1..1</td>
<td>owner_id: OBJECT_REF</td>
</tr>
<tr>
<td>1..1</td>
<td>time_created: DV_DATE_TIME</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>1..1</td>
<td>all_versions: List &lt;VERSION&lt;T&gt;&gt;</td>
</tr>
<tr>
<td>1..1</td>
<td>all_version_ids: List &lt;OBJECT_VERSION_ID&gt;</td>
</tr>
<tr>
<td>1..1</td>
<td>version_count: Integer</td>
</tr>
<tr>
<td></td>
<td>has_version_id (a_ver_id: OBJECT_VERSION_ID): Boolean require an_id /= Void</td>
</tr>
<tr>
<td></td>
<td>has_version_at_time (a_time:DV_DATE_TIME): Boolean require a_time /= Void</td>
</tr>
<tr>
<td></td>
<td>version_with_id (a_ver_id: OBJECT_VERSION_ID): VERSION&lt;T&gt; require has_version_id(a_ver_id)</td>
</tr>
<tr>
<td></td>
<td>version_at_time (a_time: DV_DATE_TIME): VERSION&lt;T&gt; require has_version_at_time(a_time)</td>
</tr>
</tbody>
</table>
### 6.4.2 VERSION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSION&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>One version within a Version container, containing data, create and commit audit trails, and optional attestations.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td><strong>uid</strong>: OBJECT_VERSION_ID</td>
<td></td>
</tr>
<tr>
<td><strong>preceding_version_id</strong>: OBJECT_VERSION_ID</td>
<td>Unique identifier of the version of which this version is a modification; Void if this is the first version.</td>
</tr>
</tbody>
</table>
### IMPORTED_VERSION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>IMPORTED_VERSION&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Versionable objects that have been copied from another location and imported into a local version container.</td>
</tr>
<tr>
<td>Inherit</td>
<td>VERSION&lt;T&gt;.</td>
</tr>
</tbody>
</table>
### 6.4.4 MERGED_VERSION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>MERGED_VERSION&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Merged Versions, recording the list of input versions to the merge in addition to standard Version attributes.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>VERSION&lt;T&gt;.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>input_version_ids:</td>
</tr>
<tr>
<td></td>
<td>1..1</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td>Input_version_ids_valid:</td>
</tr>
</tbody>
</table>

### 6.4.5 CONTRIBUTION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CONTRIBUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Documents a contribution of one or more versions added to a change-controlled repository.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>uid:</td>
</tr>
<tr>
<td>CLASS</td>
<td>CONTRIBUTION</td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
</tr>
<tr>
<td>1..1</td>
<td>versions: Set&lt;OBJECT_REF&gt;</td>
</tr>
<tr>
<td>1..1</td>
<td>audit: AUDIT_DETAILS</td>
</tr>
</tbody>
</table>

**Invariants**

- **uid_valid**: uid /= Void
- **audit_valid**: audit /= Void
- **Versions_valid**: versions /= Void and then not versions.is_empty
- **Description_valid**: audit.description /= Void
7 Resource Package

7.1 Overview

The common.resource package defines the structure and semantics of the general notion of an online resource which has been created by a human author, and consequently for which natural language is a factor. The package is illustrated in FIGURE 14.

![FIGURE 14 openehr.rm.common.resource Package](image)

7.1.1 Natural Languages and Translation

Authored resources contain natural language elements, and are therefore created in some original language, recorded in the original_language attribute of the AUTHORED_RESOURCE class. Information about translations is included in the translations attribute, which allows for one or more sets of translation details to be recorded. A resource is translated by doing the following:

- translating every language-dependent element to the new language;
- adding a new TRANSLATION_DETAILS instance to translations, containing details about the translator, organisation, quality assurance and so on.
- any further translations to language-specific elements in a instances of descendent type of AUTHORED_RESOURCE.

The languages_available function provides a complete list of languages in the resource.

7.1.2 Meta-data

What is normally considered the ‘meta-data’ of a resource, i.e. its author, date of creation, purpose, and other descriptive items, is described by the RESOURCE_DESCRIPTION and...
RESOURCE_DESCRIPTION_ITEM classes. The parts of this that are in natural language, and therefore may require translated versions, are represented in instances of the RESOURCE_DESCRIPTION_ITEM class. Thus, if a RESOURCE_DESCRIPTION has more than one RESOURCE_DESCRIPTION_ITEM, each of these should carry exactly the same information in a different natural language.

7.1.3 Revision History
When the resource is considered to be in a state where changes to it should be controlled, the is_controlled attribute is set to True, and all subsequent changes should have an audit trail recorded. Usually controlled resources would be managed in a versioned repository (e.g. implemented by CVS, Subversion or similar systems), and audit information will be stored somewhere in the repository (e.g. in version control files). The revision_history attribute defined in the AUTHOREDRESOURCE class is intended to act as a documentary copy of the revision history as known inside the repository, for the benefit of users of the resource. Given that resources in different places may well be managed in different kinds of repositories, having a copy of the revision history in a standardised form within the resource enables it to be used interoperably by authoring and other tools.

Every change to a resource committed to the relevant repository causes a new addition to the revision_history.

7.2 Class Definitions

7.2.1 AUTHOREDRESOURCE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>AUTHOREDRESOURCE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract idea of an online resource created by a human author.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 7.2.2 TRANSLATION_DETAILS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TRANSLATION_DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Class providing details of a natural language translation.</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signature</td>
</tr>
<tr>
<td>1..1</td>
<td>language: CODE_PHRASE</td>
</tr>
<tr>
<td>1..1</td>
<td>author: Hash&lt;String, String&gt;</td>
</tr>
<tr>
<td>0..1</td>
<td>accreditation: String</td>
</tr>
<tr>
<td>0..1</td>
<td>other_details: Hash&lt;String, String&gt;</td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Language_valid: language /= Void and then code_set(“languages”).has(lan-</td>
</tr>
<tr>
<td></td>
<td>guage)</td>
</tr>
<tr>
<td></td>
<td>Author_valid: author /= Void</td>
</tr>
</tbody>
</table>

### 7.2.3 RESOURCE_DESCRIPTION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>RESOURCE_DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Defines the descriptive meta-data of a resource.</td>
</tr>
</tbody>
</table>
### RESOURCE_DESCRIPTION

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>original_author:</td>
<td>Original author of this resource, with all relevant details, including organisation.</td>
</tr>
<tr>
<td></td>
<td>Hash&lt;String, String&gt;</td>
<td></td>
</tr>
<tr>
<td>0..1</td>
<td>other_contributors:</td>
<td>Other contributors to the resource, probably listed in “name &lt;email&gt;” form.</td>
</tr>
<tr>
<td></td>
<td>List&lt;String&gt;</td>
<td></td>
</tr>
<tr>
<td>1..1</td>
<td>lifecycle_state:</td>
<td>Lifecycle state of the resource, typically including states such as: initial, submitted, experimental, awaiting_approval, approved, superseded, obsolete.</td>
</tr>
<tr>
<td></td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>1..1</td>
<td>details:</td>
<td>Details of all parts of resource description that are natural language-dependent.</td>
</tr>
<tr>
<td></td>
<td>List&lt;RESOURCE_DESCRIPTION_ITEM&gt;</td>
<td></td>
</tr>
<tr>
<td>0..1</td>
<td>resource_package_uri:</td>
<td>URI of package to which this resource belongs.</td>
</tr>
<tr>
<td></td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>0..1</td>
<td>other_details:</td>
<td>Additional non language-sensitive resource meta-data, as a list of name/value pairs.</td>
</tr>
<tr>
<td></td>
<td>Hash&lt;String, String&gt;</td>
<td></td>
</tr>
<tr>
<td>0..1</td>
<td>parent_resource:</td>
<td>Reference to owning resource.</td>
</tr>
<tr>
<td></td>
<td>AUTHORED_RESOURCE</td>
<td></td>
</tr>
</tbody>
</table>

**Invariant**

- **Original_author_valid**: original_author /= Void and then not original_author.is_empty
- **Lifecycle_state_valid**: lifecycle_state /= Void and then not lifecycle_state.is_empty
- **Details_valid**: details /= Void and then not details.is_empty
- **Language_valid**: details for_all (d | parent_resource.languages_available.has(d.language))
- **Parent_resource_valid**: parent_resource /= Void implies parent_resource.description = Current
### 7.2.4 RESOURCE_DESCRIPTION_ITEM Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>RESOURCE_DESCRIPTION_ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Language-specific detail of resource description. When a resource is translated for use in another language environment, each RESOURCE_DESCRIPTION_ITEM needs to be copied and translated into the new language.</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>language: CODE_PHRASE</td>
<td>The localised language in which the items in this description item are written. Coded from openEHR Code Set “languages”.</td>
</tr>
<tr>
<td>purpose: String</td>
<td>Purpose of the resource.</td>
</tr>
<tr>
<td>keywords: List&lt;String&gt;</td>
<td>Keywords which characterise this resource, used e.g. for indexing and searching.</td>
</tr>
<tr>
<td>use: String</td>
<td>Description of the uses of the resource, i.e. contexts in which it could be used.</td>
</tr>
<tr>
<td>misuse: String</td>
<td>Description of any misuses of the resource, i.e. contexts in which it should not be used.</td>
</tr>
<tr>
<td>copyright: String</td>
<td>Optional copyright statement for the resource as a knowledge resource.</td>
</tr>
<tr>
<td>original_resource_uri: Hash&lt;String, String&gt;</td>
<td>URIs of original clinical document(s) or description of which resource is a formalisation, in the language of this description item; keyed by meaning.</td>
</tr>
<tr>
<td>other_details: Hash&lt;String, String&gt;</td>
<td>Additional language-sensitive resource metadata, as a list of name/value pairs.</td>
</tr>
</tbody>
</table>

#### Invariant

- Language_valid: language /= Void and then code_set(“languages”).has(language)
- purpose_valid: purpose /= Void and then not purpose.is_empty
- use_valid: use /= Void implies not use.is_empty
- misuse_valid: misuse /= Void implies not misuse.is_empty
- copyright_valid: copyright /= Void implies not copyright.is_empty
A References

A.1 General


A.2 European Projects


9 Deliverable 19,20,24: GEHR Architecture. GEHR Project 30/6/1995

A.3 CEN

10 ENV 13606-1 - Electronic healthcare record communication - Part 1: Extended architecture. CEN/ TC 251 Health Informatics Technical Committee.

11 ENV 13606-2 - Electronic healthcare record communication - Part 2: Domain term list. CEN/ TC 251 Health Informatics Technical Committee.

12 ENV 13606-4 - Electronic Healthcare Record Communication standard Part 4: Messages for the exchange of information. CEN/ TC 251 Health Informatics Technical Committee.

A.4 OMG

13 CORBAmed document: Person Identification Service. (March 1999). (Authors?)

14 CORBAmed document: Lexicon Query Service. (March 1999). (Authors?)
A.5 Software Engineering

15  Meyer B. *Object-oriented Software Construction*, 2nd Ed.  
    Prentice Hall 1997

16  Fowler M. *Analysis Patterns: Reusable Object Models*. Addison Wesley 1997

17  Fowler M, Scott K. *UML Distilled (2nd Ed.)*. Addison Wesley Longman 2000

A.6 Resources

REFERENCE MODEL

The openEHR Data Structures Information Model

Editors: {T Beale, S Heard}\(^1\), {D Kalra, D Lloyd}\(^2\)

Revision: 1.6

Pages: 39

---

1. Ocean Informatics Australia
2. Centre for Health Informatics and Multi-professional Education, University College London

© 2003-2006 The openEHR Foundation

The openEHR foundation

is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

**Founding Chairman**

David Ingram, Professor of Health Informatics, CHIME, University College London

**Founding Members**

Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

**email:** info@openEHR.org **web:** http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
# Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.3</td>
<td>CR-000041. Visually differentiate primitive types in openEHR documents.</td>
<td>D Lloyd</td>
<td>04 Sep 2003</td>
</tr>
<tr>
<td>1.3.2</td>
<td>CR-000013 Rename key classes - rename COMPOUND to CLUSTER to conform with CEN 13606.</td>
<td>D Kalra, T Beale</td>
<td>20 Jun 2003</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Improved heading layout, package naming. Made HISTORY.is_periodic a function.</td>
<td>T Beale, Z Tun</td>
<td>18 Mar 2003</td>
</tr>
</tbody>
</table>
### Acknowledgements

The work reported in this paper has been funded in by a number of organisations, including Ocean Informatics; The University College, London; Australia; The Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

Thanks to Grahame Grieve of Kestral Computing for general input and examples relating to History data.
# The openEHR Data Structures Information Model

**Rev 1.6**

## Table of Contents

1. **Introduction** .............................................................................. 7  
   1.1 Purpose .................................................................................. 7  
   1.2 Related Documents ............................................................... 7  
   1.3 Status .................................................................................. 7  
   1.4 Peer review ........................................................................... 7  
   1.5 Conformance ......................................................................... 8  
2. **Background** ............................................................................. 9  
   2.1 Requirements ........................................................................ 9  
   2.2 Design Principles ................................................................. 9  
3. **Overview** ............................................................................... 11  
   3.1 Instance Structures ............................................................... 11  
   3.2 Class Descriptions ................................................................. 12  
   3.2.1 DATA_STRUCTURE Class .................................................. 12  
4. **Item Structure Package** ......................................................... 13  
   4.1 Overview .............................................................................. 13  
   4.2 CEN EN13606 Encoding Rules ............................................. 14  
   4.2.1 ITEM_SINGLE ..................................................................... 14  
   4.2.2 ITEM_LIST ........................................................................ 14  
   4.2.3 ITEM_TABLE ....................................................................... 14  
   4.2.4 ITEM_TREE ........................................................................ 14  
   4.3 Class Descriptions ................................................................. 14  
   4.3.1 ITEM_STRUCTURE Class ................................................... 14  
   4.3.2 ITEM_SINGLE Class .......................................................... 14  
   4.3.3 ITEM_LIST Class .............................................................. 15  
   4.3.4 ITEM_TABLE Class ............................................................ 16  
   4.3.5 ITEM_TREE Class .............................................................. 18  
   4.4 Instance Structures ............................................................... 19  
   4.4.1 ITEM_SINGLE Instance Structure .................................... 19  
   4.4.2 ITEM_LIST Instance Structure ........................................ 19  
   4.4.3 ITEM_TABLE Instance Structure ...................................... 20  
   4.4.4 ITEM_TREE Instance Structure ....................................... 21  
5. **Representation Package** ........................................................ 23  
   5.1 Overview .............................................................................. 23  
   5.2 Class Descriptions ................................................................. 23  
   5.2.1 ITEM Class ......................................................................... 23  
   5.2.2 CLUSTER Class .................................................................. 24  
   5.2.3 ELEMENT Class ............................................................... 25  
6. **History Package** .................................................................... 27  
   6.1 Overview .............................................................................. 27  
   6.1.1 Semantics .......................................................................... 27  
   6.2.1 HISTORY<T: ITEM_STRUCTURE> Class ....................... 33  
   6.2.2 EVENT <T: ITEM_STRUCTURE> Class ......................... 34  
   6.2.3 POINT_EVENT <T: ITEM_STRUCTURE> Class ............ 35  
   6.2.4 INTERVAL_EVENT <T: ITEM_STRUCTURE> Class ...... 35
6.3 History Instance Structures ......................................................... 36
6.3.1 Single Sample ........................................................................ 36
6.3.2 5-minute Blood Pressure Averages ....................................... 36

A References .................................................................................... 37
A.1 General ......................................................................................... 37
A.2 European Projects ....................................................................... 37
A.3 CEN ................................................................................................. 37
A.4 OMG ................................................................................................. 37
A.5 Software Engineering ................................................................. 37
A.6 Resources ...................................................................................... 38
1 Introduction

1.1 Purpose
This document describes the common data structures used in openEHR reference model, including lists, tables, trees, and history, along with one possible data representation (hierarchical) which is compatible with the CEN 13606 EHCR standard.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations using openEHR;
- Academic groups using openEHR;
- 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 openEHR Modelling Guide
- The openEHR Support Information Model
- The openEHR Data Types Information Model

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

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

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

Blue text indicates sections under active development.

1.4 Peer review
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.
1.5 Conformance

Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Background

2.1 Requirements

The requirements for structured data in the EHR and other systems are essentially that low-level data can be expressed in standard structures. The structures which are commonly required are as follows:

- single values, e.g. weight, height, blood sugar;
- lists of named/numbered elements, e.g. blood test results;
- tables of values with named columns and/or named rows, e.g. visual acuity results;
- trees of values, e.g. biochemistry, microbiology results;
- histories of values, each of which takes any of the above forms, e.g. a time series of blood pressures, glucose levels, or imaging data.

2.2 Design Principles

The design principle which particularly applies to the data structure models described here is the need to provide explicit specifications for logical structures using the same generic representation, such as hierarchy. The logical structures include tables, lists, trees, and the concept of history.

Regardless of whether such structures are treated as pure presentation or as having semantic significance, there are at various reasons for explicitly including the semantics of logical structures which are represented in a generic way such as hierarchy, including:

- it is essential for interoperability that a structure such as a logical table, list or linear history be encoded into the generic representation in the same way by all senders and receivers of information, otherwise there is no guarantee that any communicating party’s software processes the structures in the intended fashion;
- software implementors can develop software which explicitly captures the logical structures as functional interfaces which are used as the only way of building such structures. Such interfaces (assuming they are bug-free) guarantee that all application software always creates correct structures - there is no need to rely on caller software each time making low level calls to create a table or list out of hierarchy elements;
- the use of a functional interface for such types means that application software at the receiver’s end can always process incoming information in its intended form, enabling correct presentation of data on the screen.

One of the motivations for defining logical data structures explicitly is to remove the ambiguity in recording structure and time in previous EHR specifications and standards, such as CEN 13606, GEHR, GEHR Australia, and HL7v3 CDA specifications. The alternative in the past was to simply use generic hierarchical structures; there was no agreement in the standard about how a table might be represented, similarly, time had no standard representation. Where single values were recorded, an attribute meaning ‘time of recording’ was set appropriately; if a time series was required, there was no clear guideline as to how to model it. One way would have been to build a double list which is logically a two column table, whose first column was time-point data, but many other approaches are possible. The standardised approach removes all such ambiguity, and improves the quality of data and software.
3 Overview

The `rm.data_structures` package contains two packages: the `item_structure` package and the `history` package. The first describes generic, path-addressable data structures, while the latter describes a generic notion of linear history, for recording events in past time. The `data_structures` package is illustrated in FIGURE 1.

![Diagram of rm.data_structures Package]

The `data_structures` package itself contains a single class, `DATA_STRUCTURE`, which is the ancestor of all openEHR data structures. Its only feature is the function `as_hierarchy`, which is implemented by each subtype of `DATA_STRUCTURE`, in order to generate a physical representation of the structure in CEN EN13606 form. The 13606 form is usually less optimal than the openEHR form, but is compatible with the less semantically rich standard, and is guaranteed (in theory) to be comprehensible to other systems which support CEN EN13606 as an interoperability standard.

3.1 Instance Structures

Diagrams of typical instances of the structures are included throughout this document. Each instance is shown in both physical and logical form. The physical form shows the instances which will occur in data if the structure is implemented using the `representation` package. The logical form shows the same instance in a logical form only - i.e. hiding the physical implementation. Only the latter form is used in other openEHR documents. In all instance diagrams, the following shorthand is used for well-known attribute names:

- “m = xxxx” - means “meaning = xxxx”, i.e. the meaning of the `archetype_node_id` attribute inherited from the `LOCATABLE` class.
- “n = xxxx” - means “name = xxxx”, i.e. the value of the `name` attribute inherited from the `LOCATABLE` class.
- “v = xxxx” - means “value = xxxx”, i.e. the value of the `value` attribute from the `ELEMENT` class.
### 3.2 Class Descriptions

#### 3.2.1 DATA_STRUCTURE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DATA_STRUCTURE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent class of all data structure types. Includes the <code>as_hierarchy</code> function which can generate the equivalent CEN EN13606 single hierarchy for each subtype’s physical representation. For example, the physical representation of an ITEM_LIST is <code>List&lt;ELEMENT&gt;</code>; its implementation of <code>as_hierarchy</code> will generate a CLUSTER containing the set of <code>ELEMENT</code> nodes from the list.</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
<tr>
<td>Function</td>
<td>Meanings</td>
</tr>
<tr>
<td><code>as_hierarchy</code>:</td>
<td>Hierarchical equivalent of the physical representation of each subtype, compatible with CEN EN 13606 structures.</td>
</tr>
<tr>
<td>Invariants</td>
<td><code>As_hierarchy_exists</code>: <code>as_hierarchy /= Void</code></td>
</tr>
</tbody>
</table>
4 Item Structure Package

4.1 Overview

The Item_Structure classes presented here are a formalisation of the need for generic, archetypable data structures, and are used by all openEHR reference models.

The subtypes of the ITEM_STRUCTURE class explicitly model the logical data structure types which typically occur in health record data, and include ITEM_SINGLE (for single values such as a patient weight), ITEM_LIST (for lists such as parts of an address), ITEM_TREE (for hierarchically structured data such as a microbiology report) and ITEM_TABLE (for tabular data such as visual acuity or reflex test results). Each of these classes defines a functional interface, has an optimal physical representation using the basic types CLUSTER and ELEMENT from the representation package, and can generate a CEN EN13606-compliant hierarchical representation of its data. Any system implementing these types is guaranteed to create data which represents the logical structures of lists, tables and trees identically.

Data values are connected to spatial structures via the value attribute of the ELEMENT class of the Representation cluster. This class also carries an important attribute null_flavor, whose value indicates how to read the value. A small domain termlist containing values such as “unknown”, “not disclosed”, “undetermined”, etc, as described in the Flavours of Null vocabulary in the openEHR Support Information Model.

The openEHR class model for spatial structures is illustrated in FIGURE 2. It should be noted that these classes (ITEM_LIST etc) are not equivalents of similarly named classes (such as List<T>) in most data structure libraries - they also include per-node name, archetype_node_id and leaf node value and null flavour, and path capabilities.

![FIGURE 2 rm.data_structures.item_structure Package](image-url)
4.2 CEN EN13606 Encoding Rules

4.2.1 ITEM_SINGLE
An ITEM_SINGLE object is encoded in EN13606 as a single ELEMENT object.

4.2.2 ITEM_LIST
An ITEM_LIST object is encoded in EN13606 as a CLUSTER object containing the set of ELEMENTs from the openEHR list.

4.2.3 ITEM_TABLE
The ITEM_TABLE encoding rules are as follows:
- A CLUSTER is required as the parent of all columns.
- Each column is represented by a CLUSTER, whose name value is the name of the column.
- Each row item in a given column is represented by an ELEMENT under the relevant column CLUSTER.
- The name of each ELEMENT object is the name of its row.

4.2.4 ITEM_TREE
Data of an ITEM_TREE instance are simply replicated as is to produce the correct EN13606 hierarchical form.

4.3 Class Descriptions

4.3.1 ITEM_STRUCTURE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM_STRUCTURE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent class of all spatial data types.</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_HIERARCHICAL_PROPOSITION</td>
</tr>
<tr>
<td>HL7v3</td>
<td>CDA Structure abstract type.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_STRUCTURE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invariants</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 ITEM_SINGLE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM_SINGLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Logical single value data structure.</td>
</tr>
</tbody>
</table>
### 4.3.3 ITEM_LIST Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Logical list data structure, where each item has a value and can be referred to by a name and a positional index in the list. The list may be empty.</td>
</tr>
<tr>
<td>Use</td>
<td>Used to represent any data which is logically a list of values, such as blood pressure, most protocols, many blood tests etc.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Not to be used for time-based lists, which should be represented with the proper temporal class, i.e. HISTORY.</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_LIST_PROPOSITION</td>
</tr>
<tr>
<td>HL7v3</td>
<td>CDA 1.0 List Entry type.</td>
</tr>
<tr>
<td>Inherit</td>
<td>ITEM_STRUCTURE</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>items: List&lt;ELEMENT&gt;</td>
<td>Physical representation of the list.</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>item_count: Integer</td>
<td>Count of all items</td>
</tr>
<tr>
<td>names: List&lt;DV_TEXT&gt;</td>
<td>Retrieve the names of all items</td>
</tr>
</tbody>
</table>
### 4.3.4 ITEM_TABLE Class

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>List&lt;CLUSTER&gt;</td>
<td>Physical representation of the table as a list of CLUSTERS, each containing the data of one column of the table.</td>
</tr>
</tbody>
</table>

**Invariants**

- **Valid_structure**: items.forall({ITEM}.type = “ELEMENT”)
- **Names_valid**: names /= Void

**CLASS** | **ITEM_TABLE**
---|---
**Purpose** | Logical table data structure, in which columns are named and ordered. Some columns may be designated ‘key’ columns, containing key data for each row, in the manner of relational tables. This allows row-naming, where each row represents a body site, a blood antigen etc. All values in a column have the same data type.

**Use** | Used to represent any data which is logically a table of values, such as blood pressure, most protocols, many blood tests etc.

**MisUse** | Not used for time-based data, which should be represented with the temporal class HISTORY. The table may be empty.

**CEN** | n/a

**GEHR** | G1_TABLE_PROPOSITION, G1_MATRIX_PROPOSITION

**HL7v3** | RIM structured types Table_structure, Table_cell, Table etc ; CDA 1.0 Table Entry type.

**Inherit** | ITEM_STRUCTURE
<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>column_names: List&lt;DV_TEXT&gt;</td>
</tr>
<tr>
<td></td>
<td>ith_row(i:Integer): List&lt;ELEMENT&gt;</td>
</tr>
<tr>
<td></td>
<td>require i &gt; 0 and i &lt; row_count</td>
</tr>
<tr>
<td></td>
<td>has_row_with_name(a_key:String): Boolean</td>
</tr>
<tr>
<td></td>
<td>require a_key /= Void and then not a_key.empty</td>
</tr>
<tr>
<td></td>
<td>has_column_with_name(a_key:String): Boolean</td>
</tr>
<tr>
<td></td>
<td>require a_key /= Void and then not a_key.empty</td>
</tr>
<tr>
<td></td>
<td>named_row(a_key:String): List&lt;ELEMENT&gt;</td>
</tr>
<tr>
<td></td>
<td>require has_row_with_name(a_key)</td>
</tr>
<tr>
<td></td>
<td>has_row_with_key(keys:Set&lt;String&gt;): Boolean</td>
</tr>
<tr>
<td></td>
<td>row_with_key(key_vals:Set&lt;String&gt;): List&lt;ELEMENT&gt;</td>
</tr>
<tr>
<td></td>
<td>require has_row_with_key(key_vals)</td>
</tr>
<tr>
<td></td>
<td>element_at_cell_ij(i, j:Integer): ELEMENT</td>
</tr>
<tr>
<td></td>
<td>require i &gt;= 1 and i &lt;= column_count</td>
</tr>
<tr>
<td></td>
<td>j &gt;= 1 and j &lt;= row_count</td>
</tr>
<tr>
<td></td>
<td>element_at_named_cell (row_key, col_key:String): ELEMENT</td>
</tr>
<tr>
<td></td>
<td>require has_row_with_name(row_key)</td>
</tr>
<tr>
<td></td>
<td>has_column_with_name(column_key)</td>
</tr>
<tr>
<td></td>
<td>as_hierarchy: CLUSTER</td>
</tr>
</tbody>
</table>
### 4.3.5 ITEM_TREE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM_TREE</th>
</tr>
</thead>
</table>
| **Invariants** | Valid_structure: items.forall({ITEM}.type = “CLUSTER” and then {ITEM}.items.forall({ITEM}.type = “ELEMENT”))  
Row_names_valid: row_names /= Void  
Column_names_valid: column_names /= Void |

#### Purpose
Logical tree data structure. The tree may be empty.

#### Use
Used to represent data which are logically a tree such as audiology results, microbiology results, biochemistry results.

#### MisUse
The CEN cluster is effectively the only data structure available in CEN, and is equivalent to the ITEM_TREE type.

#### GEHR
G1_TREE_PROPOSITION

#### HL7v3
This can be constructed with CDA 1.0 Lists. Act and Act_relationship are the closest correspondents in the HL7v3 RIM.

#### Inherit
ITEM_STRUCTURE

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1 items: LIST&lt;ITEM&gt;</td>
<td>Physical representation of the tree.</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_element_path(a_path: String): Boolean</td>
<td>True if path ‘a_path’ is a valid leaf path</td>
</tr>
<tr>
<td>element_at_path(a_path: String):ELEMENT require has_element_path(a_path)</td>
<td>Return the leaf element at the path ‘a_path’</td>
</tr>
<tr>
<td>as_hierarchy: CLUSTER</td>
<td>Generate a CEN EN13606-compatible hierarchy, which is the same as the tree’s physical representation.</td>
</tr>
</tbody>
</table>
4.4 Instance Structures

4.4.1 ITEM_SINGLE Instance Structure

FIGURE 3 illustrates a ITEM_SINGLE instance, in both physical and logical forms.

![Diagram of ITEM_SINGLE instance structure with physical and logical forms.](image)

4.4.2 ITEM_LIST Instance Structure

FIGURE 4 illustrates a typical ITEM_LIST structure, in this case for a BP protocol.

![Diagram of ITEM_LIST instance structure with physical and logical forms.](image)
4.4.3 ITEM_TABLE Instance Structure

FIGURE 5 illustrates a table of visual acuity test results.

Physical Form

Logical Form

FIGURE 5 ITEM_TABLE Instance Structure
4.4.4 ITEM_TREE Instance Structure

FIGURE 6 illustrates the logical and physical form of an example ITEM_TREE instance, representing a biochemistry result.

**Physical Form**

```
ITEM_TREE
   m = “biochemistry”
   n = “biochemistry”

ELEMENT
   m = “sample”
   n = “sample”
   value

DV_CODED_TEXT
   v = “serum”

ELEMENT
   m = “battery group”
   n = “lipid studies”

CLUSTER
   m = “battery item”
   n = “total cholesterol”
   value
   DV_QUANTITY
   v = XX

ELEMENT
   m = “battery item”
   n = “LDL cholesterol”
   value
   DV_QUANTITY
   v = XX

ELEMENT
   m = “battery item”
   n = “HDL cholesterol”
   value
   DV_TEXT
   v = “xxxxxxx”

ELEMENT
   m = “comment”
   n = “comment”
```

**Logical Form**

```
ITEM_TREE
   m = “biochemistry”
   n = “biochemistry”

m = “sample”
   n = “sample”
   v = “serum”

m = “battery group”
   n = “lipid studies”

m = “battery item”
   n = “total cholesterol”
   v = XXX

m = “battery item”
   n = “total cholesterol”
   v = XXX

m = “battery item”
   n = “total cholesterol”
   v = XXX

m = “comment”
   n = “comment”
   v = “xxxx”
```

**FIGURE 6 ITEM_TREE Instance Structure**
5 Representation Package

5.1 Overview

This package contains classes for a simple hierarchical representation of any data structure, as shown in FIGURE 7. These classes are compatible with the CEN EN13606 classes of the same names, and instances can be losslessly generated to and from EN13606 instances structures.

![Diagram](image)

FIGURE 7 rm.data_structures.representation Package

5.2 Class Descriptions

5.2.1 ITEM Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ITEM (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The abstract parent of CLUSTER and ELEMENT representation classes.</td>
</tr>
<tr>
<td>CEN</td>
<td>ITEM class</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::Observation</td>
</tr>
<tr>
<td>Synapses</td>
<td>Item class</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_HIERARCHICAL_ITEM</td>
</tr>
<tr>
<td>HL7v3</td>
<td>n/a</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 5.2.2 CLUSTER Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The grouping variant of ITEM, which may contain further instances of ITEM, in an ordered list.</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::CompositeObservation</td>
</tr>
<tr>
<td>Synapses</td>
<td>Compound class</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_HIERARCHICAL_GROUP</td>
</tr>
<tr>
<td>HL7v3</td>
<td>Act_context</td>
</tr>
<tr>
<td>Inherit</td>
<td>ITEM</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>items: List&lt;ITEM&gt;</td>
<td>Ordered list of items - CLUSTER or ELEMENT objects - under this CLUSTER.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Items_non_empty: items /= Void and then not items.empty</td>
<td></td>
</tr>
</tbody>
</table>
## 5.2.3 ELEMENT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ELEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>The leaf variant of ITEM, to which a DATA_VALUE instance is attached.</td>
</tr>
<tr>
<td>CEN</td>
<td>DataItem class</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::AtomicObservation</td>
</tr>
<tr>
<td>Synapses</td>
<td>Element class</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_HIERARCHICAL_VALUE</td>
</tr>
<tr>
<td>HL7v3</td>
<td>Act</td>
</tr>
<tr>
<td>Inherit</td>
<td>ITEM</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1 (cond) value: DATA_VALUE</td>
<td>data value of this leaf</td>
</tr>
<tr>
<td>0..1 (cond) null_flavor: DV_CODED_TEXT</td>
<td>flavour of null value, e.g. indeterminate, not asked etc</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 is_null: Boolean</td>
<td>True if value logically not known, e.g. if indeterminate, not asked etc.</td>
</tr>
</tbody>
</table>

### Invariants

- Is_null_valid: is_null = (value = Void)
- Null_flavor_indicated: is_null xor null_flavour /= Void
- Null_flavour_valid: is_null implies terminology(“openehr”).codes_for_group_name(“null flavour”, “en”).has(null_flavor.defining_code)
6 History Package

6.1 Overview

The history package defines classes which formalise the concept of past, linear time, via which historical data of any structural complexity can be recorded. It supports both instantaneous and interval event samples within periodic and aperiodic series. Data recorded in interval events are tagged with a mathematical function, including ‘point-in-time’, ‘mean’, ‘delta’ and so on, as defined by the openEHR “event math function” vocabulary. It also supports the inclusion of “summary” data, i.e. a textual or image item which summarises in some way the entire history.

Regardless of whether the actual data consist of a single sample or many, they are represented in the same way: as a history of events, i.e. as a time series, allowing all software to access data in a uniform way, whether it is a single measurement of weight, a long series of three- or four-dimensional images, or even a series of encapsulated multimedia items.

The model defines the constrained generic (otherwise known as ‘template’ or ‘parameterised’) types `HISTORY<T>, EVENT<T>, POINT_EVENT<T>, and INTERVAL_EVENT<T>` where the type parameter is constrained to the `ITEM_STRUCTURE` type, and defines the type of the data recorded in an instance of `HISTORY`. The effect is that repeated instances of spatially complex data can recur in time, corresponding to the way data are actually measured. An aperiodic series of `POINT_EVENT` instances would typically be used to represent manual measurements repeated in time. Periodic histories of `INTERVAL_EVENT` instances would typically be used to represent vital signs monitor output (which is usually delivered in averaged form potentially with additional minimum and maximum values).

As with all other parts of the openEHR reference model, the History package is designed for archetyping; archetypes define the domain semantics of `HISTORY` instances. The history package is shown in FIGURE 8.

6.1.1 Semantics

Basic Semantics

The intention of the History model is to represent time-based data for which every sample in the series is a measurement of the same phenomenon (e.g. patient heart rate) and is obtained using the same measurement method (e.g. pulse oximeter). Samples taken in this way can be reliably treated as representing changes in a phenomenon over time, and accordingly can be safely used for time-based computation, such as graphing, statistical analysis and so on. A History can contain any mixture of `POINT_EVENT` and `INTERVAL_EVENT` instances. Clearly it is impossible for the model to guarantee completely correct usage on its own, however there two major safeguards.

Firstly, the use of generic types forces the type of the data in each Event to be the same. A History of type `HISTORY<ITEM_LIST>` therefore constrains the type of the data at each Event (`EVENT.item`) to be of type `ITEM_LIST` and nothing else.

Secondly, the use of archetyping (typically within openEHR Observation archetypes) ensures the actual structure of the `ITEM_STRUCTURE` subtype is defined in the same way for every sample - e.g. a two-item list representing systolic and diastolic blood pressure.

Timing

An instance of the `HISTORY` class contains the `origin: DV_DATE_TIME` attribute, indicating what is considered the ‘0-point’ of the time series, and a series of instances of the `EVENT` subtype, each containing a `time: DV_DATE_TIME` attribute representing the absolute time of the event. The relative off-
set of any Event is computed as the difference between the EVENT.time and HISTORY.origin by the EVENT.offset function. For Interval events (i.e. instances of INTERVAL_EVENT), the time attribute always refers to the end time of the event, since this is the time at which the data (e.g. average) are true.

The origin time of a History does not have to be the time of the first sample - it might be the time of an event such as child-birth with respect to which the samples are recorded, e.g. Apgar\(^1\) scores at 1 and 3 minute offsets. Periodicity and aperiodicity are expressed via the is_periodic and period attributes. For a periodic time-series, period is set to the period duration value and is_periodic returns True. The total duration of the History is given by the HISTORY.duration function. FIGURE 9 illustrates a number of variations in History periodicity and Event type.

**Point Events**

The simplest kind of Event in a History is a “point” event, expressed by instances of the class POINT_EVENT, representing an instantaneous value. A History instance may be composed solely of Point events, as would be the case with a number of blood pressure values measured over time as the patient changes position. An Apgar result is a typical example of aperiodic point data, typically consisting of 2 or 3 events, each containing 5 values and a 6th value representing the Apgar score for that time point. Point data may also be available from monitoring devices. For fine-granularity (e.g. 1 second) data, the number of samples may be too voluminous for the health record, and more efficient

---

1. A 0-10 score indicating the health of a newborn based on breathing, heart rate, colour, muscle tone and reflexes
Interval Events

Instances of the INTERVAL_EVENT class are used to express values corresponding to an interval in time. The INTERVAL_EVENT.width attribute defines the duration of the interval; and the inherited time value corresponds to the trailing edge of the event.

The meaning of an Interval event in this model is that the values effectively summarise actual instantaneous values of a datum that have occurred during the period of the event interval. The mathematical meaning of the data of any particular interval event is given by the math_function attribute. This is coded by the openEHR Terminology group “event math function”, and takes values such as “minimum”, “maximum”, “average”, “delta” and so on. The particular math functions used in each Interval event in a History may vary throughout the History; for example, one 4-hour Interval event might contain data representing average values, while a following event might contain data representing maximum values. Such data can be conveniently used for generating sophisticated graphs of the underlying datum over time. FIGURE 11 illustrates a History containing 2 pairs of 4-hour blood pres-
sure Interval events, with each pair containing maximum and mean blood pressure value structures for +4h and +8h timepoints (each of which consist of a systolic and diastolic value).

**FIGURE 11** Structure of HISTORY<T> of INTERVAL_EVENTs

Interval events can overlap other interval or point events within the same History. A common situation where this occurs is with measurement of different periods of vital signs, such as 4-hourly blood pressure events, overlapped by a 24-hour event which contains the values over a period of 6 x 4 hour periods. In general a long Interval event can overlap any combination of Point or Interval events, as shown in **FIGURE 12**.

**FIGURE 12** Overlapping Events

**Change Data**

One subcategory of interval data that deserves mention is change data. There are three event math function terms used for indicating changes in data values as follows:

- “change”: this means that the value recorded is the difference between the value now and the value some time previously. It can be positive or negative;
- “increase”: the value recorded for the change is positive. The name (i.e. `ELEMENT.name`) chosen for the item in an archetype carries the semantic of positivity e.g. "increase of ....; rise of....; ....gain" etc;
The following examples show how the data and these math functions are coordinated.

- weight last week was 76 kg. Wait this week = 74 kg. Possible instances:

<table>
<thead>
<tr>
<th>Item Name in Archetype</th>
<th>Value stored</th>
<th>Type</th>
<th>Math Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>“weight change”</td>
<td>+2kg</td>
<td>DV_QUANTITY</td>
<td>“change”</td>
</tr>
<tr>
<td>“weight loss”</td>
<td>(+)2kg</td>
<td>DV_QUANTITY</td>
<td>“decrease”</td>
</tr>
<tr>
<td>“weight loss”</td>
<td>True</td>
<td>Boolean</td>
<td>“decrease”</td>
</tr>
</tbody>
</table>

- weight last week was 80 kg. Weight this week = 83 kg. Possible instances:

<table>
<thead>
<tr>
<th>Item Name in Archetype</th>
<th>Value stored</th>
<th>Type</th>
<th>Math Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>“weight change”</td>
<td>(+)3kg</td>
<td>DV_QUANTITY</td>
<td>“change”</td>
</tr>
<tr>
<td>“weight increase”</td>
<td>(+)3kg</td>
<td>DV_QUANTITY</td>
<td>“increase”</td>
</tr>
<tr>
<td>“weight gain”</td>
<td>True</td>
<td>Boolean</td>
<td>“decrease”</td>
</tr>
</tbody>
</table>

The use of these math function indicators allows the correct representation of change values, no matter how they were captured, in a computable form.

**Summary Event Data**

A relatively common situation particularly in laboratory testing is the existence of a “summarising” event which accompanies more detailed historical data. Examples where this arises include:

- a series of exams with a single radiologist report for all of them (the report might include one or more key images);
- graphical summary of a dynamic challenge test such as Glucose tolerance test;
- some comment about the pattern of values on a set of observed values in series.

Such data are accommodated within the model via the optional `HISTORY.summary` attribute, which is itself a structure, archetypable separately from the structure of the main data. In the first example above, the summary data might consist of an `ITEM_SINGLE` object containing a textual report; in the second, an `ITEM_SINGLE` object containing a image within a `DV_MULTIMEDIA` instance.

**Efficient Representation of Fine-grained Device Data**

A useful practical consequence of the of Interval Events is that it allows long periods of relatively stable data to be represented with a single Interval event, while interesting perturbations will be represented with a number of fine-grained Interval or Point Events. In the example in FIGURE 13 5 Event instances are used represent 4 hours of data consisting to 14,400 x 1 second samples from a blood pressure monitor. The optional `INTERVAL_EVENT.sample_count` attribute can be used to record the number of original samples summarised in the event. In the illustration, the `math_function` is shown as mean; clearly in the first long period, the monitored datum was not absolutely flat. The implication is that the recording software was configured to regard variations in a small band (e.g. 5mm Hg) as insignificant, and only to create new Event objects when the underlying values moved outside the band. Another approach would have been to create two Interval Event objects for each long period,

---

1. Examples provided by G Grieve of Kestral Computing, Australia
one giving minimum value, the other maximum value, still based on the principle of generating one such pair for periods when the underlying data remained within specified limits. Regardless of the details, this general approach provides a way to include hours of fine-grained data from devices like vital signs monitors in very little space; the data simply need to be transformed into equivalent openEHR History form first.

State

A feature particular to a model of recording historical data for scientific and clinical use is the ability to record ‘state’. In openEHR, ‘state’ is understood as information pertinent to the correct interpretation of the primary data. A simple example is where the primary datum is heartrate; useful state data would be the level of exersion of the subject (resting, after 3 minutes cycling etc). In clinical recording sitatoins, the state data is often crucial to the safe use of the primary data, since the latter might be normal or abnormal depending on the patient state. In openEHR there are two ways of recording state. One is via the use of a separate HISTORY structure within the OBSERVATION class (see ehr.composition.content.entry package). The other is via the use of the state attribute of type ITEM_STRUCTURE defined in the class EVENT itself. Experience with openEHR archetypes and systems has shown that the latter method corresponds to the most common clinical need, which is to be able to record the state at the time of the event (the other method allows for the recording of independent state events). A simple example is the recording of 3 glucose levels during a glucose tolerance test. The state information for each event is, respectively (in a typical test):

- 0-minute sample: “post 8-hour fast”;
- 1-hour sample: “post 75g oral glucose challenge”;
- 2-hour sample: “post 75g oral glucose challenge”.

FIGURE 13 Data compression effect of History with Interval Events
The History structure for this example is illustrated in FIGURE 14.

![FIGURE 14 State in HISTORY]

6.2 Class Descriptions

6.2.1 HISTORY<T: ITEM_STRUCTURE> Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>HISTORY&lt;T: ITEM_STRUCTURE&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Root object of a linear history, i.e. time series structure.</td>
</tr>
<tr>
<td>CEN</td>
<td>Time was encoded as part of the Item structure.</td>
</tr>
<tr>
<td>GEHR</td>
<td>Time was encoded as part of the G1_HIERARCHICAL_PROPOSITION structure.</td>
</tr>
<tr>
<td>HL7v3</td>
<td>The data type HIST&lt;T&gt; is equivalent in intention to HISTORY&lt;T&gt; for histories of primitive values; for histories of complex data, Act/Act-relationship structures have to be used.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_STRUCTURE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>origin : DV_DATE_TIME</td>
<td>Time origin of this event history. The first event is not necessarily at the origin point.</td>
</tr>
<tr>
<td>1..1</td>
<td>events: List &lt;EVENT&lt;T&gt;&gt;</td>
<td>The events in the series.</td>
</tr>
<tr>
<td>0..1</td>
<td>period: DV_DURATION</td>
<td>Period between samples in this segment if periodic.</td>
</tr>
</tbody>
</table>
### 6.2.2 EVENT <T: ITEM_STRUCTURE> Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>HISTORY&lt;T: ITEM_STRUCTURE&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>duration: DV_DURATION</td>
</tr>
<tr>
<td>0..1</td>
<td>summary: ITEM_STRUCTURE</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_periodic: Boolean</td>
<td>Indicates whether history is periodic.</td>
</tr>
<tr>
<td>as_hierarchy: CLUSTER</td>
<td>Generate a CEN EN13606-compatible hierarchy of the physical representation.</td>
</tr>
</tbody>
</table>

#### Invariants

- origin_exists: origin /= Void
- events_exists: (events /= Void and then not events.is_empty) xor summary /= Void
- periodic_validity: is_periodic xor period = Void

---

### 6.2.2 EVENT <T: ITEM_STRUCTURE> Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EVENT &lt;T: ITEM_STRUCTURE&gt; (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Defines the abstract notion of a single event in a series. This class is generic, allowing types to be generated which are locked to particular spatial types, such as EVENT&lt;ITEM_LIST&gt;. Subtypes express point or interval data.</td>
</tr>
<tr>
<td>HL7v3</td>
<td>The data type HistoryItem HXIT&lt;T&gt; is close to EVENT&lt;T&gt; in its intent.</td>
</tr>
<tr>
<td>Inherit</td>
<td>LOCATABLE</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>time: DV_DATE_TIME</td>
<td>Time of this event. If the width is non-zero, it is the time point of the trailing edge of the event.</td>
</tr>
<tr>
<td>data: T</td>
<td>The data of this event.</td>
</tr>
<tr>
<td>state: ITEM_STRUCTURE</td>
<td>Optional state data for this event.</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent: HISTORY&lt;T&gt;</td>
<td>Redefinition of LOCATABLE.parent to be of type HISTORY.</td>
</tr>
<tr>
<td>offset: DV_DURATION</td>
<td>Offset of this event from origin.</td>
</tr>
</tbody>
</table>
6.2.3 POINT_EVENT <T: ITEM_STRUCTURE> Class

CLASS

EVENT <T: ITEM_STRUCTURE> (abstract)

Invariants

Time_exists: \( \text{time} \neq \text{Void} \)
Data_exists: \( \text{data} \neq \text{Void} \)
Offset_validity: \( \text{offset} \neq \text{Void} \) and then \( \text{offset} = \text{time} - \text{parent.origin} \)

6.2.4 INTERVAL_EVENT <T: ITEM_STRUCTURE> Class

CLASS

INTERVAL_EVENT <T: ITEM_STRUCTURE>

Purpose

Defines a single point event in a series.

Inherit

EVENT

Attributes

Signature | Meaning
---|---
Invariants

Optional count of original samples to which this event corresponds.

Functions

Signature | Meaning
---|---

Invariants

Width_valid: \( \text{width} \neq \text{Void} \)
Math_function_validity: terminology("openehr").codes_for_group_name("event math function", "en").has(math_function.defining_code)
Interval_start_time_valid: \( \text{interval_start_time} \neq \text{Void} \)

CLASS

POINT_EVENT <T: ITEM_STRUCTURE>

Purpose

Defines a single point event in a series.

Inherit

EVENT

Attributes

Signature | Meaning
---|---
Invariants

CLASS

INTERVAL_EVENT <T: ITEM_STRUCTURE>

Purpose

Defines a single interval event in a series.

Inherit

EVENT

Attributes

Signature | Meaning
---|---
Invariants

CLASS

EVENT <T: ITEM_STRUCTURE> (abstract)
6.3 History Instance Structures

6.3.1 Single Sample
FIGURE 15 illustrates a single weight measurement. The Event objects contain the timing information, which in this case is simply the time of measurement (the origin).

6.3.2 5-minute Blood Pressure Averages
FIGURE 16 illustrates two Interval events representing 5 minute blood pressure averages, the first at 5 minutes’ offset from an initial event and lasting 5 minutes, the second 5 minutes later.
A References

A.1 General

A.2 European Projects

A.3 CEN

A.4 OMG
11. CORBAmed document: *Person Identification Service*. (March 1999). (Authors?)

A.5 Software Engineering
14. Fowler M. *Analysis Patterns: Reusable Object Models*. Addison Wesley 1997
Fowler M, Scott K. *UML Distilled (2nd Ed.)*. Addison Wesley Longman 2000

**A.6 Resources**

END OF DOCUMENT
REFERENCE MODEL

The openEHR Data Types Information Model

Editors: {T Beale, S Heard}¹, {D Kalra, D Lloyd}²

Revision: 2.0

Pages: 91

1. Ocean Informatics Australia
2. Centre for Health Informatics and Multi-professional Education, University College London

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>CR-000176. Make DV_QUANTIFIED accuracy optional.</td>
<td>S Heard</td>
<td>01 Feb 2005</td>
</tr>
<tr>
<td></td>
<td>CR-000163. 163: Add identifiers to FEEDER_AUDIT for originating and gateway systems.</td>
<td>H Frankel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000121. Improve DV_EHR_URI model to support Xpath style paths.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000161. Support distributed versioning. Remove functions from DV_EHR_URI.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.96</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9.1</td>
<td>Improve implementation guidance. DV_ORDINAL.limits type corrected to REFERENCE_RANGE&lt;DV_ORDINAL&gt;.</td>
<td>D Lloyd</td>
<td>22 Feb 2005</td>
</tr>
<tr>
<td>1.9</td>
<td>CR-000126. Correct details of partial date/time classes.</td>
<td>T Beale</td>
<td>09 Dec 2004</td>
</tr>
<tr>
<td></td>
<td>CR-000112. Add DV_PARTIAL_DATE_TIME class</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000113. Add DATA_VALUE subtype for identifying real-world entities</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000118. Make package names lower case.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000119. Improve Data types documentation.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000102. Make DV_TEXT language and charset optional.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>CR-000023. TERM_MAPPING.match should be coded/enumerated.</td>
<td>G Grieve</td>
<td>09 Mar 2004</td>
</tr>
<tr>
<td></td>
<td>CR-000069. Correct date/time types statistical descriptions.</td>
<td>A Goodchild</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000046. Rename COORDINATED_TERM to CODE_PHRASE</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and DV_CODED_TEXT.definition to defining_code.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000084. Rename DV_COUNTABLE to DV_COUNT.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000090. Make TERM_MAPPING.purpose optional.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000091. Correct anomalies in use of CODE_PHRASE and DV_CODED_TEXT.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000094. Add lifecycle state attribute to VERSION; correct DV_STATE.</td>
<td>DSTC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000095. Remove property attribute from Quantity package.</td>
<td>DSTC, S Heard</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Formally validated using ISE Eiffel 5.4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7.9</td>
<td>CR-000066. Make DV_ORDERED.normal_range a function. Correct UML for DV_QUANTITY.</td>
<td>Z Tun</td>
<td>10 Nov 2003</td>
</tr>
<tr>
<td>1.7.8</td>
<td>CR-000053. Make DV_ORDINAL.limits a function.</td>
<td>T Beale</td>
<td>02 Nov 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000054. Move DV_QUANTIFIED.is_normal to DV_ORDERED</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000055. Redefine DV_ORDERED.less_than as infix function '&lt;'.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>1.7.4</td>
<td>CR-000020. Move VERSION.charset to DV_TEXT, territory to TRANSACTION. Remove VERSION.language.</td>
<td>A Goodchild</td>
<td>10 Jun 2003</td>
</tr>
<tr>
<td>1.7.3</td>
<td>DV_INTERVAL now inherits from INTERVAL to avoid duplicating semantics. (Formally validated).</td>
<td>T Beale</td>
<td>25 Mar 2003</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Moved definitions and assumed types to Support Reference Model. No semantic changes.</td>
<td>T Beale</td>
<td>25 Feb 2003</td>
</tr>
<tr>
<td>1.6.1</td>
<td>Rome CEN TC 251 meeting. Updates to HL7 comparison text. DV_DATE now inherits from DV_CUSTOMARY_QUANTITY.</td>
<td>S Heard, T Beale</td>
<td>27 Jan 2003</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>1.6</td>
<td><strong>Sam Heard complete review.</strong> Changed constant terminology of DV_PHYSICAL_DATA. Added new chapter for generic implementation guidelines, and a new section for assumed types. Post-conditions moved to invariants: DV_TEXT.value, DV_ORDERED.is_simple, DV_PARTIAL_DATE.probable_date, possible_dates, DV_PARTIAL_TIME.probable_time, possible_times. Minor updates to HL7 comparison text. Added explanation to HL7 section.</td>
<td>S Heard, T Beale</td>
<td>13 Dec 2002</td>
</tr>
<tr>
<td>1.5.9</td>
<td>Minor corrections: DV_ENCAPSULATED; DV_QUANTITY.units defined to be String; changed COORDINATED_TERM class (but semantically equivalent).</td>
<td>T Beale</td>
<td>10 Nov 2002</td>
</tr>
<tr>
<td>1.5.8</td>
<td>Changed name of LINK package to URI. Major update to Text cluster classes and explanation. Updated HL7 data type comparison.</td>
<td>T Beale, D Kalra, D Lloyd, M Darlison</td>
<td>1 Nov 2002</td>
</tr>
<tr>
<td>1.5.7</td>
<td>DV_TEXT_LIST reverted to TEXT_LIST. DV_LINK no longer a data type; renamed to LINK and moved to Common RM. Link package renamed to “URI”.</td>
<td>S Heard, Z Tun, T Beale, D Kalra, M Darlison</td>
<td>18 Oct 2002</td>
</tr>
<tr>
<td>1.5.6</td>
<td>Rewrite of TIME_SPECIFICATION parse specs. Adjustments to DV_ORDINAL.</td>
<td>T Beale</td>
<td>16 Sep 2002</td>
</tr>
<tr>
<td>1.5.5</td>
<td>Timezone not allowed on pure DV_DATE in ISO8601.</td>
<td>T Beale, S Heard</td>
<td>2 Sep 2002</td>
</tr>
<tr>
<td>1.5.4</td>
<td>Moved DV_QUANTIFIED.units and property attributes to DV_QUANTITY. Introduced DV_WORLD_TIME.to_quantity. Added fractional_second to DV_TIME, DV_DATE_TIME, DV_DURATION.</td>
<td>T Beale, S Heard</td>
<td>29 Aug 2002</td>
</tr>
<tr>
<td>1.5.3</td>
<td>Further corrections - removed derived ‘/’ markers; renamed TERM_MAPPING.granularity to match. Improved explanation of DV_ORDINAL. DV_QUANTIFIED.units is now a DV_PARSABLE. REFERENCE_RANGE.meaning is now a DV_TEXT. DV_ENCAPSULATED.uri is now a DV_URI. DV_LINK.type is now a DV_TEXT. Detailed review by Zar Zar Tun (DSTC).</td>
<td>T Beale, S Heard, P Schloeffél, D Lloyd, Z Tun</td>
<td>20 Aug 2002</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Further corrections - removed derived ‘/’ markers; renamed TERM_MAPPING.granularity to match.</td>
<td>T Beale, D Lloyd, S Heard</td>
<td>15 Aug 2002</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Minor corrections.</td>
<td>T Beale, S Heard</td>
<td>15 Aug 2002</td>
</tr>
<tr>
<td>1.5</td>
<td>Rewrite of section describing text types; addition of new attribute DV_CODED_TEXT.mappings. Removal of TERM_REFERENCE.concept_code.</td>
<td>T Beale, S Heard</td>
<td>1 Aug 2002</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Minor changes to text. Corrections to DV_CODED_TEXT relationships. Made DV_INTERVAL.lower_unbounded and DV_INTERVAL.upper_unbounded functions.</td>
<td>T Beale, Z Tun</td>
<td>16 Jul 2002</td>
</tr>
</tbody>
</table>
The openEHR Data Types Information Model

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.2</td>
<td>$DV_LINK$.meaning changed to $DV_TEXT$ (typo in table). Added abstract class $DV_WORLD_TIME$.</td>
<td>T Beale, D Lloyd</td>
<td>14 Jul 2002</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Changes to $DV_ENCAPSULATED$, $DV_PARSABLE$ invariants.</td>
<td>T Beale, Z Tun</td>
<td>10 Jul 2002</td>
</tr>
<tr>
<td>1.4</td>
<td>$DV_ENCAPSULATED$.text_equivalent renamed to $DV_ENCAPSULATED$.alternate_text. Added invariant for QUANTITY.precision.</td>
<td>T Beale, D Lloyd</td>
<td>1 Jul 2002</td>
</tr>
<tr>
<td>1.3</td>
<td>Added timezone to $DV_TIME$ and $DV_DATE_TIME$ and sign to $DV_DURATION$; added linguistic_order to TERM_RELATION; added as_display_string and as_canonical_string to all types. Added $DV_STATE$.is_terminal. Renamed TERM_TEXT as CODED_TEXT.</td>
<td>T Beale, D Lloyd</td>
<td>30 Jun 2002</td>
</tr>
<tr>
<td>1.2</td>
<td>Minor corrections to Text package.</td>
<td>T Beale</td>
<td>15 May 2002</td>
</tr>
<tr>
<td>1.1</td>
<td>Numerous small changes, including: term equivalents, relationships and quantity reference ranges.</td>
<td>T Beale, D Lloyd, D Kalra, S Heard</td>
<td>10 May 2002</td>
</tr>
<tr>
<td>1.0</td>
<td>Separated from the openEHR Reference Model.</td>
<td>T Beale</td>
<td>5 May 2002</td>
</tr>
</tbody>
</table>

Acknowledgements

The work reported in this paper has been funded by a number of organisations, including The University College, London; The Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia; Ocean Informatics, Australia.
Table of Contents

Copyright Notice ................................................................................................... 2
Amendment Record ................................................................................................ 3
Acknowledgements ................................................................................................. 6
Table of Contents .................................................................................................... 7

1 Introduction ........................................................................................................... 11
  1.1 Purpose ............................................................................................................ 11
  1.2 Related Documents ....................................................................................... 11
  1.3 Status ............................................................................................................. 11
  1.4 Peer review .................................................................................................... 11
  1.5 Conformance ................................................................................................ 11

2 Background .......................................................................................................... 13
  2.1 Scope ............................................................................................................. 13
  2.2 Design Criteria ............................................................................................ 13
  2.3 Prior Work .................................................................................................... 14

3 Introduction .......................................................................................................... 15
  3.1 Overview ....................................................................................................... 15
  3.2 Package Structure ......................................................................................... 15

4 Basic Package ...................................................................................................... 16
  4.1 Overview ....................................................................................................... 16
  4.1.1 Requirements ........................................................................................... 16
  4.2 Class Descriptions ....................................................................................... 17
    4.2.1 DATA_VALUE Class ............................................................................. 17
    4.2.2 DV_BOOLEAN Class .......................................................................... 18
    4.2.3 DV_STATE Class ................................................................................ 18
    4.2.4 DV_IDENTIFIER Class ........................................................................ 19

5 Text Package ....................................................................................................... 21
  5.1 Overview ....................................................................................................... 21
    5.1.1 Requirements ......................................................................................... 21
      5.1.1.1 Narrative Text ................................................................................... 22
      5.1.1.2 Terminological Entities ..................................................................... 22
    5.1.2 Design ................................................................................................... 23
    5.1.3 Qualification .......................................................................................... 24
    5.1.4 Meaning Modification ......................................................................... 24
      5.1.4.1 Mode-changing Terms ................................................................. 24
      5.1.4.2 Context Sensitivity .......................................................................... 24
      5.1.4.3 Negation ........................................................................................ 25
      5.1.4.4 Representation of Meaning-Modifying Terms .............................. 25
    5.1.5 Mappings ............................................................................................. 26
      5.1.5.1 Classification (Broader Terms) ....................................................... 26
      5.1.5.2 Equivalent / Synonymous Terms .................................................. 27
      5.1.5.3 More Specific Mappings (Narrower Terms) ................................... 27
      5.1.5.4 The Unified Medical Language System (UMLS) ......................... 27
      5.1.5.5 Legacy Mapping Scenarios ......................................................... 28
    5.1.6 Language Translations .......................................................................... 28
  5.2 Class Descriptions ......................................................................................... 28
    5.2.1 DV_TEXT Class .................................................................................... 28
5.2.2 TERM_MAPPING Class .............................................................. 30
5.2.3 CODE_PHRASE Class ................................................................. 31
5.2.4 DV_CODED_TEXT Class ........................................................... 32
5.2.5 DV_PARAGRAPH Class ............................................................ 32

6 Quantity Package ................................................................... 35
6.1 Overview ............................................................................................. 35
6.1.1 Requirements ................................................................................. 36
6.1.2 Design............................................................................................ 38
6.2 Class Descriptions ............................................................................... 39
6.2.1 DV_ORDERED Class ................................................................... 40
6.2.2 DV_INTERVAL<T : DV_ORDERED> Class ..................................... 41
6.2.3 REFERENCE_RANGE<T:DV_ORDERED> Class .................... 42
6.2.4 DV_ORDINAL Class ................................................................. 42
6.2.5 DV_QUANTIFIED Class ............................................................. 43
6.2.6 DV_MEASURABLE Class .......................................................... 45
6.2.7 DV_QUANTITY Class ................................................................. 45
6.2.8 DV_COUNT Class ........................................................................ 46
6.2.9 Units Syntax .................................................................................. 47
6.2.10 DV_QUANTITY_RATIO Class................................................... 48
6.2.11 DV_CUSTOMARY_QUANTITY Class...................................... 48

7 Date Time Package ................................................................ 50
7.1 Overview ............................................................................................. 50
7.1.1 Requirements................................................................................. 50
7.1.2 Design............................................................................................ 51
7.2 Definitions ........................................................................................... 53
7.3 Class Descriptions ............................................................................... 54
7.3.1 DV_WORLD_TIME Class........................................................... 54
7.3.2 DV_DATE Class ........................................................................... 54
7.3.3 DV_TIME Class............................................................................ 55
7.3.4 DV_DATE_TIME Class............................................................... 56
7.3.5 DV_DURATION Class................................................................. 58
7.3.6 DV_PARTIAL_DATE Class......................................................... 59
7.3.7 DV_PARTIAL_TIME Class......................................................... 60
7.3.8 DV_PARTIAL_DATE_TIME Class ............................................. 61

8 Time_specification Package .................................................. 63
8.1 Overview ............................................................................................. 63
8.1.1 Requirements................................................................................. 63
8.1.2 Design............................................................................................ 64
8.2 Class Descriptions ............................................................................... 64
8.2.1 DV_TIME_SPECIFICATION Class............................................. 64
8.2.2 DV_PERIODIC_TIME_SPECIFICATION Class ........................ 65
8.2.2.1 Phase-linked Time Specification Syntax .................................................... 65
8.2.2.2 Event-linked Periodic Time Specification Syntax ....................... 66
8.2.3 DV_GENERAL_TIME_SPECIFICATION Class..................... 66
8.2.3.1 General Time Specification Syntax ..................................................... 67

9 Encapsulated Package ........................................................... 69
9.1 Overview ............................................................................................. 69
1 Introduction

1.1 Purpose
This document describes the openEHR Data Types Information Model, used throughout the openEHR Reference Model. The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations developing EHR systems;
- Academic groups studying the 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 openEHR Modelling Guide
- The openEHR Support Information Model

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

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

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

Blue text indicates sections under active development.

1.4 Peer review
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.

1.5 Conformance
Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Background

2.1 Scope

The data type specification presented here defines the data types which are used in the openEHR reference models. Harmonisation of data types between information models used by related services in a health information infrastructure is essential to reducing the conversion work and potential for errors between these services. Accordingly, the openEHR data type specification is intended to work not only for the EHR, but also for other models defined by openEHR, such as the openEHR demographic and terminological models.

The types described here have been derived from data types used in the GEHR [14], Synapses and SynEx [10], CEN 13606 [11], [13] and particularly the HL7v3 [15] reference models.

2.2 Design Criteria

Over and above the need to satisfy the functional requirements of clinical data, three concerns have driven the design of the openEHR data types:

1. clarity of expression
2. ease of implementation
3. interoperability with data types from other standards

The first of these has led to models which try to clearly convey the semantics of types required by the clinical domain. The use of constraints (pre- and post-conditions and class invariants) and a comprehensible class structure ensures formal self-consistency, correct type-substitutability and implementability in object-oriented formalisms. Types have been designed so as not to clash with norms of object-oriented languages and libraries, in particular, class names and the inbuilt types. Accordingly, all types presented here have a logical name commencing with ‘DV_’, ensuring that there is no clash with a type in the implementation formalism, hence the type DV_DATE presented here will not be confused with the type DATE which appears in many programming languages or libraries.

Object-oriented languages which have been considered include IDL, C++, Java, C#, Eiffel, Delphi and Python. Each of these languages obeys some variant of the well-known semantics of classes, encapsulation, typing and inheritance. The data types described here follow the tenets of object-orientation defined in UML most closely, while being careful not to invalidate their implementation in any language. The models have all been validated by implementation in the Eiffel language, the closest available semantic fit for UML, and currently the most powerful of mainstream object-oriented formalisms.

Implementability in XML-schema has also been an important design criterion, and the current data types remove many of the problems which the GEHR and CEN data types presented for XML-schema. It should be noted at the outset that there has been no attempt to support XML-DTD, since it has no type system, and cannot reliably be reasoned about in an object-oriented way.

To simplify implementation in all object-oriented formalisms, including IDL, programming languages and X-schema, multiple inheritance has been avoided (only marginal cases were identified anyway). Generic classes have however been used extensively, since they significantly clarify the model. Type genericity is available in Eiffel, C++, and soon in Java; for languages not having it, there is a well-known transformation from models containing generic classes to classes for non-generic types systems (see for example [3]).
Implementability in relational databases has also been considered, and appears relatively straightforward, since only the data view of the types needs to be represented. Many implementations are likely to use only a single String or XML string to represent each entire data instance, which significantly simplifies things.

2.3 Prior Work

Four other type systems for clinical data, namely the GEHR data types, the HL7 v3 data types, the CEN 13606 data item types, and the Corbamed data types were carefully scrutinised in order to ensure a) that all needed types were covered in the openEHR specification, and b) that data conversion will be possible. Concepts from all three are cross-referenced throughout this specification where possible.

Because the HL7v3 data type specification is a widely available and comprehensive specification for clinician data types, particular attention has been paid to incorporating its semantics, as well as fixing errors or shortcomings. While there are differences both in design approach and in detail, a significant debt must be recognised to the authors of this work, from which many ideas in the present specification were drawn. A detailed discussion is found under Comparison with HL7v3 Types on page 81.
3 Introduction

3.1 Overview

This data type specification constitutes the lowest level of technical specification of openEHR, and describes a set of types suitable for use in clinical and related information structures. In order for such types to exist, a set of primitive types is assumed, namely Integer, Real, Boolean, Character, String, List<T>, Set<T>, and Array<T>. These have standard definitions in the OMG object model used in UML, OCL, and are available in almost all type systems. The exact assumptions are described in the openEHR Support Reference Model. A number of symbolic definitions (similar to constants in programming) are also described in the Support RM.

The data types described here are named with the class prefix “DV_”, and inherit from the class DATA_VALUE. They have two distinct uses in reference models. Firstly, they may be used as “data values” in reference model structures wherever the DATA_VALUE class appears, for example, in the EHR Reference Model via the ELEMENT.value attribute. Additionally, specific subtypes of the data types described here can also be used as attribute types in other classes in reference models, such as date/times, coded terms and so on. The difference is that in the former case, only subtypes of DATA_VALUE may be used, whilst in the latter case, other types may be used as well, from the assumed set of basic types.

3.2 Package Structure

The package structure of the openEHR data types is illustrated in FIGURE 1.

![FIGURE 1 openehr.rm.data_types Package](image-url)
4 Basic Package

4.1 Overview

The data_types.basic package, illustrated in FIGURE 2, contains types for the concepts of bi-state, state (in a state machine) and real-world entity identifiers (see the openEHR Common IM for a discussion on identifier types).

4.1.1 Requirements

**Bi-state Values**

One of the most basic types of data is boolean or bi-state data. The need here is for a type which both includes a boolean value, and which inherits from the type DATA_VALUE, enabling it to be used as an ELEMENT.value.

**State Machine States**

A type is required to represent state values of a state machine. In a simple system of data types, a simple integer would appear sufficient to perform this job. However, in an archetyped framework, a distinct type is required, so that it can be archetyped not by the constraints used for integers, but by a state machine definition instead. The type DV_STATE is provided for this purpose. An example of a state machine which models the lifecycle of a medication order is illustrated in FIGURE 3. This definition would appear in an archetype; the values of a DV_STATE object are then restricted to the values of the states in the definition.

**Real-world Entity Identification**

Real world entities (RWEs) such as people, car engines, invoices, and appointments all have identifiers. Although many of these are designed to be unique within a jurisdiction or issuing space, they are often not, due to data entry errors, bad design (ids which are too small or incorporate some non-unique characteristic of the identified entities), bad process (e.g. non-synchronised id issuing points); identity theft (e.g. via theft of documents of proof or hacking). In general, while some real world identifiers (RWIs) are “nearly unique”, none can be guaranteed so.

Examples of RWE identifiers which are intended to be unique within the space of the issuing authority or organisation include:
• driver’s licence id
• social security number
• passport number
• prescription id

The defining characteristic of many RWE ids is that they continue to identify the entities in question, regardless of how they change in time; for example a social security number does not change when someone changes their hair colour or even their gender (both of which attributes may be recorded in the database). There may be a general principle whereby any RWE id in fact doesn’t identify an individual entity so much as its passage in time and space.

In general it should be the case that if two RWE ids are equal, they refer to the same RWE. RWE identifiers differ from information entity (IE) identifiers in that they are not assigned by the computing infrastructure, but by organisations or people.

The model defined here in the DV_IDENTIFIER class allows the recording of three things as part of identifying an item of interest:

• the issuing authority of the kind of id used (e.g. it might be the federal department of health);
• the assigner of the id to the item being identified. This is usually the organisation which created the item being identified;
• the identifier given to the item of interest.

In addition, the type of item being identified can also be recorded.

4.2 Class Descriptions

4.2.1 DATA_VALUE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DATA_VALUE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Serves as a common ancestor of all data value types in openEHR models.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.1 - 3.13</td>
</tr>
</tbody>
</table>
### 4.2.2 DV_BOOLEAN Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DATA_VALUE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEN</td>
<td>The Data_Item class in CEN is a mixture of DATA_VALUE and ELEMENT in openEHR.</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::ObservationValue</td>
</tr>
<tr>
<td>HL7</td>
<td>DataValue (ANY)</td>
</tr>
</tbody>
</table>

#### Invariants

The **DV_BOOLEAN** class should not be used as a replacement for naively modelled enumerated types such as male/female etc. Such values should be coded, and in any case the enumeration often has more than two values.

#### ISO 18308

(none)

#### CEN

Not provided as a subtype of Data Item

#### Synapses

A special use of the Numeric class is defined to represent the Boolean data type, limiting the permitted values to zero or one.

#### HL7

Boolean (BL) type. HL7 also allows NULL values, which is problematic for a) the understanding of Boolean, normally expected to be either True or False, and b) implementation in nearly all programming languages, in which Boolean is a value type (i.e. cannot be NULL).

#### Inherit

DATA_VALUE

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: Boolean</td>
<td>Boolean value of this item.</td>
</tr>
</tbody>
</table>

#### Invariants

Value_exists: value /= Void

### 4.2.3 DV_STATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>For representing state values which obey a defined state machine, such as a variable representing the states of an instruction or care process.</td>
</tr>
</tbody>
</table>
### 4.2.4 DV_IDENTIFIER Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_IDENTIFIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Type for representing identifiers of real-world entities. Typical identifiers include drivers licence number, social security number, vertans affairs number, prescription id, order id, and so on.</td>
</tr>
<tr>
<td>Use</td>
<td>DV_IDENTIFIER is used to represent any identifier of a real thing, issued by some authority or agency.</td>
</tr>
<tr>
<td>Misuse</td>
<td>DV_IDENTIFIER should not be used to express identifiers generated by the infrastructure to information items; the types OBJECT_ID and OBJECT_REF and subtypes are for this purpose.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>(none)</td>
</tr>
<tr>
<td>GEHR</td>
<td>GEHR had a type PHYSICAL_DATA for the purpose of recording locations of items, such as specimen bottles.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>issuer: String</td>
<td>Authority which issues the kind of id used in the id field of this object.</td>
</tr>
<tr>
<td>assigner: String</td>
<td>Organisation that assigned the id to the item being identified.</td>
</tr>
<tr>
<td>id: String</td>
<td>The identifier value. Often structured, according to the definition of the issuing authority’s rules.</td>
</tr>
<tr>
<td>type: String</td>
<td>The identifier type, such as “prescription”, or “SSN”. One day a controlled vocabulary might be possible for this.</td>
</tr>
</tbody>
</table>

**Invariants**

- `issuer_valid: issuer /= Void and then not issuer.is_empty`
- `assigner_valid: issuer /= Void and then not assigner.is_empty`
- `id_valid: id /= Void and then not id.is_empty`
- `type_valid: type /= Void and then not type.is_empty`
5 Text Package

5.1 Overview

The data_types.text package contains classes for representing all textual values in the health record, including plain text, coded terms, and narrative text. It is illustrated in FIGURE 4.

5.1.1 Requirements

The sections below describe the requirements of text data types. Two overriding principles should be noted at the outset with regard to text.

1. Regardless of what terminologies are (or are not) available to the clinician and/or the software, the primary requirement is that in all cases clinicians are able to record exactly what they want to say. This means that if they want to record something very general, such as “cold” or a very specific term such as “Ross River Virus infection” they should be able to, whether or not the appropriate terms are available. However, the facility should be available to additionally ‘code’ any such textual item, at the time or indeed at some later time, so as to satisfy reporting or other needs.

2. It is assumed that any client of terminology, such as the EHR, uses a terminology service which provides a complete interface to the terminology part of the knowledge domain. The coded text type reflects this. Accordingly, there is no concept of “post-coordination” allowed by the data types described here: the only thing that is available from the terminology service is a key which refers to a lexical entity, which may be a single term or a code phrase, and which may be part of a reference terminology and/or linked to element(s) of underlying ontologies. It is also assumed that there is no direct access to any particular terminology; access to all terminologies (whether simple coded lexicons or large semantic networks) is via the same abstract interface.

Terminology Ids are likely to be of various types.

1. Terminology_Id = “LOCAL”: this constant value means that the origin of allowable values is described within the archetype. This is coded to allow translation. The local archetype then only needs the set of codes and the local translation. The archetype may contain a translation table if required.
2. Terminology_Id = “[authority]:[Domain value set]”. This might be “HL7:Gender”

3. Terminology_Id=”100178” - a unique value in the accepted set of terminologies from an authoritative source. These MUST be universally known.

The set of legal Terminology IDs will need to be tightly controlled by an authoritative agency such as HL7 or openEHR.

5.1.1.1 Narrative Text

Narrative text items are used in the EHR in a number of cases, including:

- values of structural attributes in the reference model;
- recording of subjective or imprecise patient responses, particularly quantities or dates not deemed sufficiently precise to be represented using structured quantitative or date/time date types;
- recording of narrative statements, e.g. visual observations;
- recording of tracts of prose, e.g. overall findings and conclusions, prognoses;
- recording of values which would normally be coded, but for which no code and/or no terminology service is available.

While narrative text items themselves are not themselves coded, they may have code phrases associated with them, as described below under Mappings, and may be mixed within a paragraph with coded items.

5.1.1.2 Terminological Entities

Textual entities available in a terminology service are used in the health record to enable processing, from simple queries to decision support. Reasons for using terminology include:

- To guarantee interoperability of meaning. For instance, if the term “cold” is recorded in plain text, it could be interpreted as “feeling cold”, “C.O.L.D” (chronic obstructive lung disease), “rhinorrhoea”, “coryza” or “U.R.T.I. (upper respiratory tract infection), among others. If, however, it is coded from a terminology such as ICD10 or SNOMED-CT, any party reading the data (including software) knows the intention, since the meaning of the code in the terminology is unambiguous.

- To standardise textual renderings of terms and avoid informal shorthand. For example, practitioners wanting to write “systolic blood pressure” write things like “systolic BP”, “systolic bp”, “sys. BP.” and so on; use of coded terms ensures that such abbreviations are either avoided, or associated with an unambiguous meaning.

- For unambiguous naming of problems, medications or diagnoses for support of knowledge-based tools such as prescribing packages and other decision support applications.

- For standardised names of things in the record e.g. a heading of “Physical examination” or an entry such as “Differential diagnosis”.

- For finite sets of values (“value sets”), e.g. Blood Group = ‘A|B|AB|O’.

- For classifying other data for the purpose of statistical studies, e.g. by putting ICD disease group classifiers on actual disease names entered in health records.

A basic requirement for interoperability of text items, coded using terms (i.e. where the text is the official rubric for the code), is that both the rubric and the code (or ‘code-phrase’) must be recorded, to ensure the originally intended text is retained for receivers of EHR information who do not have access to the same terminologies used at the origin (or indeed any terminology service at all). However, where a terminology service is available, the key can be used to unambiguously locate the string
value of the term, and can also be used to find translations in other languages. (Note that these comments do not apply to mappings, which are described below).

In modern terminologies, there are semantic networks of links emanating from most coded terms, which classify them or relate them to other terms. Such links provide a means for decision support to make inferences about specific things found in the record. For instance if the term “leukaemia” is found, queries to the terminology service can be made in order to deduce that the patient has both a “cancer” and a “disease of the immune system” (assuming leukaemia is classified under these more general terms in the terminology).

This specification assumes the existence of a terminology service which is responsible for interrogating actual terminologies and performing validated coordination of terms, i.e. creating combinations deemed valid by the underlying source terminology, potentially without even assigning a new code to the result. All validated coordination is carried out inside the terminology service, and any “term” made available by the service is already “coordinated” (one might now think of such terms as having been “pre-coordinated” by the terminology service itself, even though they are not pre-coordinated inside any given terminology). For example, the coordination “foot, left” (a shorthand way of writing the relationship “foot has-laterality left”) could be created by the terminology service from the source terms “foot”, “left” and “has-laterality” from a terminology such as SNOMED. Any such coordination must be valid within the source terminology, i.e. correspond to valid relationships defined therein.

The class `DV_CODED_TEXT` described here captures the association of two things:

- the code phrase of a code phrase provided by the terminology service, recorded in the `defining_code` attribute.
- the text rubric of the code phrase, recorded in the `value` attribute (inherited from `DV_TEXT`);

The class `CODE_PHRASE` records a key, in the form of arguments to some retrieval function in the terminology service interface.

There are different semantics attached to different coordinations of terms. Two main categories of coordination have been described in the literature: “qualification” and “modification”. A common definition of the first is that “qualification narrows meaning” - i.e. creates a new term whose possible real world instances are within the set denoted by the original root term. Modification on the other hand changes the meaning of a root term. Various cases are described below under Meaning Modification. Both coordination types are managed by the terminology service.

Coded terms may also be mapped to terms from other terminologies, which may be intended as equivalents, classifiers, or something in between. The section below on Mappings deals with these.

### 5.1.2 Design

All atomic text items are either instances of the type `DV_TEXT` or of `DV_CODED_TEXT`. The former allows the expression of text with optional formatting and hyperlinking. The latter connects the text value to a key in the terminology service, with the implication that the key refers to a terminological entity lexically and semantically identical to the text value.

The model of `DV_CODED_TEXT` is designed to capture the actual coded term chosen by the user or software at runtime; it is implicitly assumed that this includes whichever synonym (term of equivalent meaning from the same terminology) was chosen, for terminologies supporting synonyms, and any coordination of underlying distinct terms. A `DV_CODED_TEXT` instance can only be used if the final textual value chosen by the user is lexically identical to the rubric returned by the terminology.
service for the key; if the user makes even the slightest change, the identity of rubric / key is lost, and a mapping (see Mappings on page 26) should be used instead.

The type **DV_TEXT** should be used wherever a coded or non-coded text item is allowed, while the type **DV_CODED_TEXT** should be used wherever a text item must be coded.

The type **DV_PARAGRAPH** allows larger tracts of text to be built up from lists of **DV_TEXT** instances, i.e. instances of **DV_TEXT** and **DV_CODED_TEXT**, as illustrated in FIGURE 5.

5.1.3 Qualification
Qualification is the process of making a term more specific through the post-coordination of additional terms. It occurs when a terminology defines relationships between a primary term and other terms that qualify the primary. For example a coordination using the term “bronchitis” which creates a qualified term might be “acute bronchitis”; all real world instances of the latter are also instances of the former.

5.1.4 Meaning Modification
Terms that change the meaning of other terms are often known as “modifiers”. The difference between modification and qualification is that modifiers change the meaning so that the modified term as a whole does not refer to instances of the unmodified term. We describe below the particular types of modifiers and how they are represented using the text data types.

5.1.4.1 Mode-changing Terms
One class of modifiers is exemplified by the addition of words like “risk of”, “fear of”, “history of” and so on. These are sometimes called *mode-changing* terms, since they change the “mode” of the root term from the present to the past (“history of”), a potential future (“risk of”) or some other alternate reality. Terms which are modified in this way should never be matched in queries searching for the root term; for example, a query for “coronary diease” (of the patient) should not match “family history of coronary disease”.

5.1.4.2 Context Sensitivity
There are many terms whose meaning is changed by the context in which they are stated, such as within a certain kind of note or test result. Consider the following:

- a blood sugar level after a 75gm oral loading has a different meaning than a fasting blood sugar;
- a systolic blood pressure in the pulmonary artery has a different meaning than a systemic arterial blood pressure;
- “total hip replacement” in the context of a “planned procedure”;
- “meningitis” in the context of a “differential diagnosis”.

**FIGURE 5** A DV_PARAGRAPH
5.1.4.3 Negation

Negation is a special kind of mode change and has been a serious design challenge in the past, because modifiers like “not” or “no” only make sense when attached to some terms, and create nonsensical values or ambiguities by arbitrarily association with other terms.

5.1.4.4 Representation of Meaning-Modifying Terms

Rather than provide explicit features for representing modifier terms within DV_CODED_TEXT, the general principle underlying representation of all post-coordinations other than qualifications, is that a higher-level, archetyped structure such as an ENTRY (defined in the EHR RM), is a minimal indivisible unit of information. Such higher-level entities can have internal structure, and it is possible (and desirable) to achieve the effect of combinations of terms through this structure. In the case of ENTRY, it will be via structuring of CLUSTER/ELEMENT objects. The general rule is: to obtain the full meaning of any terms found in the record, all of the node names in any ENTRY (coded or not) must be considered from the root to the relevant leaf. Conversely, the “final” meaning of any term in the record cannot be known in isolation from the rest of the terms in the structure.

Accordingly, the concept “family history of coronary disease” is represented as an ENTRY whose root is named (for example) “subject family history”, and which includes further structure, which may be in greater of lesser detail; the coded term “coronary disease” would appear somewhere in this structure. The actual structure is completely defined by appropriate archetypes. Contrary to some perceptions, there is no general way to represent concepts such as “family history of coronary disease”, since it will vary depending on how much detail is recorded. Where some GPs routinely record just the simplest form, others may record the details of which family members had heart problems and exactly what they were.

The same approach is used for context-dependent terms. Archetypes defining contexts such as “planned procedures” or “differential diagnosis” will use these terms as their root nodes; as a result, the meaning of any term appearing below the root can only be understood by including the root. Once again, the exact structures are completely dependent upon archetyping, and may be simple or quite sophisticated.

Negations are more complex than might first be apparent and are best handled by good archetype design. Terminologies might provide a term such as “No known allergies” which is helpful. But if someone has an allergy of some sort, the medicolegal requirement might be to record that the person has no known allergies to penicillin or another class of medication that is being prescribed. The often-proposed approach of using a generic negation ‘modifier’ to deal with such issues results in further problems. Consider the use of negation with liver - “no liver”, “no palpable liver”, “no liver disease”, “no history of liver disease”, “no liver function”, “no liver function tests”. The meaning of negated terms may be nonsensical and difficult to interpret.

A basic principle of dealing with negatives is to realise that most naive suggested use cases are quite ambiguous as stated. Does “no allergies” mean “no reported episode of allergy”, “no allergic reactions ever”, “no known allergies to medication” or something else? Does it mean that these statements are taken as given by the patient, or determined by tests? Like all medical phenomena, allergies must be described in some detail for the EHR to be of any real use. Almost inevitably, this precludes the use of negated terms. Since the actual information structure will be determined in advance by archetype designers, clinicians will almost never be in the situation of having to negate a term. However, if the need does arise, it should be dealt with by a negative or quantitative answer, i.e. a value rather than a name. For example, in any ENTRY describing current problems, the clinician may record the name/value pair “allergies: NONE”. Here, “allergies” will be a DV_CODED_TEXT, and “NONE” will
be either a `DV_CODED_TEXT` or a `DV_TEXT`; the two will be associated by a containing object, such as an instance of the `ELEMENT` class from the EHR RM.

## 5.1.5 Mappings

In a number of circumstances, both plain text and coded text items are mapped to terms from other terminologies. In theory, this should never occur, since it means that relationships between terms which should only be knowable in the knowledge base (in the form of the terminology service, or something else) are being created and transmitted as part of EHR information, potentially invalidating or overriding the knowledge base. Where mappings are required, the proper approach is to create thesauri within the knowledge environment, and map through them. Unfortunately, in some cases, activities in the real world do not respect the information/knowledge boundary, hence the model described here includes an explicit `mapping` concept, which itself includes a “purpose” and a “match” indicator. Matching corresponds to the categories described below.

### 5.1.5.1 Classification (Broader Terms)

Any text item, whether coded or not, may be classified with a coded term, for research, reporting and decision support purposes. For example, a GP working in tropical Australia may wish to write “Ross River infection”, and be working with ICD9, which does not contain this term (although ICD9-CM does). He or she will use a plain text item, but will still be able to map it to an ICD9 classifier, such as the code for “arbovirus infection NOS”. The same approach can be used for adding a classifying term to a coded text item. The utility of classifier terms is various: they allow decision support to make more powerful inferences; in situations where the available terminologies do not provide the classification inbuilt, and where it is known that not all users of EHR data will have terminologies available. In data terms, classification mapping can be visualised as illustrated in FIGURE 6.

![FIGURE 6 Plain Text and Coded Text with Classifier(s)](image)

Classifying mappings are represented by adding a term to the `mappings` list of the original term. Each mapping is explicitly represented with an instance of `TERM_MAPPING` which indicates both the term being associated with the original text item, and a value of ‘>’ for the `match` attribute, which indicates that the mapping is “broader”. The possible values of the `match` attribute are ‘>’ (broader), ‘<’ (narrower).
5.1.5.2 Equivalent / Synonymous Terms

Data from pathology laboratories has often been coded using a terminology local to the laboratory, due to lack of or economic unfeasibility of using existing widespread terminologies for the job. However, some laboratories also supply a nearest equivalent code from a well-known terminology such as LOINC, to enable the receiver of the data to process it in a more standard fashion. Here, “equivalence” is taken to mean a term of the same meaning but from a different vocabulary.

Another instance where equivalent terms might be supplied is to effect the translation of terms across specialist vocabularies such as nursing vocabularies when sharing EHRs across jurisdictions.

In theory, the cleanest way for senders and receivers of data coded with both a local and a more standard equivalent to deal with the mapping problem is for the originator of the local terminology to provide a complete thesaurus of translations into one or more recognised terminologies. However, in practice, laboratories using the HL7 v2.x messaging standard usually encode a primary term and equivalents with the HL7 CE data type, meaning that equivalents are included only with the term they are used with. A similar pragmatic approach to mapping equivalent terms in the EHR is likely to be used with the data types described here, and can be effected with the same mapping approach as for classification.

A further situation in which text values - this time plain text - is mapped to equivalent terms is when natural language processing is used to generate coded terms for existing free-text prose. The aim of such processing is to detect word phrases and associate them with a coded term of the same meaning, without obliterating the original text. In this case, an instance of DV_CODED_TEXT is associated with an instance of DV_TEXT via the mappings attribute.

In all cases with equivalents, the value of the match attribute is ‘=’, indicating that the mapping is a synonym.

5.1.5.3 More Specific Mappings (Narrower Terms)

Occasionally, there is a need to create a mapping to a term of narrower meaning than the original text item. Circumstances in which this occurs include when a clinician wants to record a syndrome such as “croup” or “influenza”, but the terminology does not contain these general terms, although it does contain more specific terms, e.g. “viral laryngo-tracheitis” or “influenza type A”. Clearly the clinician should be allowed to record what he/she wants (as plain text if necessary), but it should also be possible to add a mapping to the more precise term. For mappings to narrower terms, the value of the match attribute is ‘<’.

5.1.5.4 The Unified Medical Language System (UMLS)

It has been argued in GEHR [14] that UMLS reference terms should also be supplied with occurrences of coded terms, in the form of the UMLS concept unique identifier, or “CUI”. UMLS is a way of encoding terms developed at the National Library of Medicine in the United States, and consists of a meta-thesaurus, in which terms from any extant term set (such as ICD, SNOMED, READ) can be cross-referenced. UMLS CUIs could turn out to be extremely useful for decision support and reporting.

The proper use of UMLS is that terms from particular terminologies are passed to a UMLS interface and a CUI + rubric received in response. However, the mapping approach described above could also be used to map UMLS CUIs to existing text or terms in an EHR; in this case, a DV_CODED_TEXT is constructed for each UMLS “term”, where the code is the CUI and the rubric is the text rendering of
the CUI (guaranteed unique in UMLS). The same approach can be used for any other thesaurus which becomes available in the future.

5.1.5.5 Legacy Mapping Scenarios
In cases where legacy data has to be converted to openEHR-compliant data, and only codes are available, e.g. ICD or ICPC codes, the following approach is recommended:

- create a new `DV_TEXT` whose value is “(not available)”
- add a mapping to the `DV_TEXT`, with:
  - `purpose` = “legacy conversion”
  - `match` = “=”
  - `target` = CODE_PHRASE object whose `code_string` and `terminology_id` are set to correspond to the available code in the legacy data.

This expresses the reality that no text was ever recorded in the legacy system; rather a code was recorded directly in the data field. In the converted data, this code is more correctly considered a mapping.

5.1.6 Language Translations
There does not appear to be any argument for recording language translations in the text data types (i.e. other than the primary language version in use by the EHR server, which of course might not be English). Firstly, there is no way to know which ones might be needed, since it is not known in advance to whom the data might be sent; if it were included on this basis, one would presumably have to include all available translations. Secondly, the availability of language translations is very mixed, and in some cases quite limited. Even large terminologies like SNOMED support only a few languages, or translations for particular subsets. If extra translations were included with each term, the selection would be quite arbitrary, and for small terminologies, there might be none. Lastly, the intended use of language translation for coded terms is that it occur at the receiver’s end, via the use of the same terminology in another language mode.

5.2 Class Descriptions

5.2.1 `DV_TEXT` Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th><code>DV_TEXT</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A plain text item, which may contain any amount of legal characters arranged as e.g. words, sentences etc (i.e. one <code>DV_TEXT</code> may be more than one word). Any <code>DV_TEXT</code> may be “coded” by adding mappings to it.</td>
</tr>
<tr>
<td>Use</td>
<td>Fragments of text, whether coded or not are used on their own as values, or to make up larger tracts of text which may be marked up in some way, eventually going to make up paragraphs.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 2.6, 2.9</td>
</tr>
<tr>
<td>Synapses</td>
<td>The Text data value class can contain either plain text or a term taken from a terminology system (coding scheme).</td>
</tr>
<tr>
<td>CLASS</td>
<td>DV_TEXT</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
</tr>
<tr>
<td>HL7</td>
<td>Roughly equivalent to CWE (coded with extensions) - i.e. a text value which may optionally be coded.</td>
</tr>
</tbody>
</table>

**Inherit** DATA_VALUE

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: String</td>
<td>Displayable rendition of the item, regardless of its underlying structure. For DV_CODED_TEXT, this is the rubric of the complete term as provided by the terminology service. No carriage returns, line feeds, or other non-printing characters permitted.</td>
</tr>
<tr>
<td>mappings: List <code>&lt;TERM_MAPPING&gt;</code></td>
<td>terms from other terminologies most closely matching this term, typically used where the originator (e.g. pathology lab) of information uses a local terminology but also supplies one or more equivalents from well-known terminologies (e.g. LOINC).</td>
</tr>
<tr>
<td>formatting: String</td>
<td>A format string of the form “name:value; name:value...”, e.g. &quot;font-weight : bold; font-family : Arial; font-size : 12pt;&quot;. Values taken from W3C CSS2 properties lists “background” and “font”.</td>
</tr>
<tr>
<td>hyperlink: DV_URI</td>
<td>Optional link sitting behind a section of plain text or coded term item.</td>
</tr>
<tr>
<td>language: CODE_PHRASE</td>
<td>Optional indicator of the localised language in which the value is written. Coded from openEHR Code Set “languages”. Only used when either the text object is in a different language from the enclosing ENTRY, or else the text object is being used outside of an ENTRY or other enclosing structure which indicates the language.</td>
</tr>
<tr>
<td>encoding: CODE_PHRASE</td>
<td>Name of character set in which this value is encoded. Coded from openEHR Code Set “character sets”.</td>
</tr>
</tbody>
</table>

**Invariants**

- **Value_valid**: value /= void and then not value.is_empty and then not (value.has(CR) or value.has(LF))
- **Language_valid**: language /= Void implies code_set(“languages”).has(language)
- **Encoding_valid**: encoding /= Void implies code_set(“character sets”).has(encoding)
- **Mappings_valid**: mappings /= void implies not mappings.is_empty
- **Formatting_valid**: formatting /= void implies not formatting.is_empty
5.2.2 TERM_MAPPING Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TERM_MAPPING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents a coded term mapped to a DV_TEXT, and the relative match of the target term with respect to the mapped item. Plain or coded text items may appear in the EHR for which one or mappings in alternative terminologies are required. Mappings are only used to enable computer processing, so they can only be instances of DV_CODED_TEXT.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for adding classification terms (e.g. adding ICD classifiers to SNOMED descriptive terms), or mapping into equivalents in other terminologies (e.g. across nursing vocabularies).</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 4.5</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>target: CODE_PHRASE</td>
<td>The target term of the mapping.</td>
</tr>
<tr>
<td>match: Character</td>
<td>The relative match of the target term with respect to the mapped text item. Result meanings:</td>
</tr>
<tr>
<td></td>
<td>• ‘&gt;’: the mapping is to a broader term e.g. original text = “arbovirus infection”, target = “viral infection”</td>
</tr>
<tr>
<td></td>
<td>• ‘=’: the mapping is to a (supposedly) equivalent to the original item</td>
</tr>
<tr>
<td></td>
<td>• ‘&lt;’: the mapping is to a narrower term e.g. original text = “diabetes”, mapping = “diabetes mellitus”.</td>
</tr>
<tr>
<td></td>
<td>• ‘?’: the kind of mapping is unknown.</td>
</tr>
<tr>
<td>purpose: DV_CODED_TEXT</td>
<td>Purpose of the mapping e.g. “automated data mining”, “billing”, “interoperability”</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>narrower: Boolean</td>
<td>The mapping is to a narrower term.</td>
</tr>
<tr>
<td>ensure</td>
<td>match = ‘&lt;’ implies Result</td>
</tr>
</tbody>
</table>
### 5.2.3 CODE_PHRASE Class

**Purpose**

A fully coordinated (i.e. all “coordination” has been performed) term from a terminology service (as distinct from a particular terminology).

**ISO 18308**

STR 4.2

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminology_id: TERMINOLOGY_ID</td>
<td>Identifier of the distinct terminology from which the code_string (or its elements) was extracted.</td>
</tr>
<tr>
<td>code_string: String</td>
<td>The key used by the terminology service to identify a concept or coordination of concepts. This string is most likely parsable inside the terminology service, but nothing can be assumed about its syntax outside that context.</td>
</tr>
</tbody>
</table>

**Invariants**

Target_exists: target /= Void

**Purpose_valid:** purpose /= Void implies terminology(“openehr”).codes_for_group_name(“term mapping purpose”, “en”).has(purpose.defining_code)

Match_valid: is_valid_match_code(match)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target_exists:</td>
<td>target /= Void</td>
</tr>
<tr>
<td>Purpose_valid:</td>
<td>purpose /= Void implies terminology(“openehr”).codes_for_group_name(“term mapping purpose”, “en”).has(purpose.defining_code)</td>
</tr>
<tr>
<td>Match_valid:</td>
<td>is_valid_match_code(match)</td>
</tr>
</tbody>
</table>

The equivalent model class

`equivalent: Boolean`  

*ensure*

match = ‘=’  

*implies* Result

The mapping is to an equivalent term.

The broader model class

`broader: Boolean`  

*ensure*

match = ‘>’  

*implies* Result

The mapping is to a broader term.

The unknown model class

`unknown: Boolean`  

*ensure*

match = ‘?’  

*implies* Result

The kind of mapping is unknown.

The `is_valid_match_code` function

`is_valid_match_code(c: Character): Boolean`  

*ensure*

Result := c = ‘>’ or c = ‘=’ or c = ‘<’ or c = ‘?’  

True if match valid.
### 5.2.4 DV_CODED_TEXT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_CODED_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A text item whose <em>value</em> must be the rubric from a controlled terminology, the key (i.e. the ‘code’) of which is the <strong>defining_code</strong> attribute. In other words: a <strong>DV_CODED_TEXT</strong> is a combination of a <strong>CODE_PHRASE</strong> (effectively a code) and the rubric of that term, from a terminology service, in the language in which the data was authored.</td>
</tr>
<tr>
<td>Use</td>
<td>Since <strong>DV_CODED_TEXT</strong> is a subtype of <strong>DV_TEXT</strong>, it can be used in place of it, effectively allowing the type <strong>DV_TEXT</strong> to mean “a text item, which may optionally be coded”.</td>
</tr>
<tr>
<td>Misuse</td>
<td>If the intention is to represent a term code attached in some way to a fragment of plain text, <strong>DV_CODED_TEXT</strong> should not be used; instead use a <strong>DVTexto</strong>nd a <strong>TERM_MAPPING</strong> to a <strong>CODE_PHRASE</strong>.</td>
</tr>
</tbody>
</table>

| ISO 18308 | STR 4.1, 4.2, 4.3 |
| CEN | Text |
| OMG HDTF | COAS::CodedElement, LooselyCodedElement. |
| Synapses | Text |
| GEHR | G1_TERM_TEXT |
| HL7 | ConceptDescriptor (CD), CodedValue (CV) and CodedSimple (CS) |
| Inherit | **DV_TEXT** |

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td><strong>defining_code</strong>:CODE_PHRASE</td>
<td>The term which the ‘value’ attribute is the textual rendition (i.e. rubric) of.</td>
</tr>
</tbody>
</table>

| Invariants | **Definition_exists**: defining_code /= Void |

### 5.2.5 DV_PARAGRAPH Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_PARAGRAPH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A logical composite text value consisting of a series of <strong>DV_TEXTs</strong>, i.e. plain text (optionally coded) potentially with simple formatting, to form a larger tract of prose, which may be interpreted for display purposes as a paragraph.</td>
</tr>
<tr>
<td>Use</td>
<td><strong>DV_PARAGRAPH</strong> is the standard way for constructing longer text items in summaries, reports and so on.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 2.6</td>
</tr>
<tr>
<td>CLASS</td>
<td>DV_PARAGRAPH</td>
</tr>
<tr>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_PARAGRAPH</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_VALUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>items: List&lt;DV_TEXT&gt;</td>
<td>Items making up the paragraph, each of which is a text item (which may have its own formatting, and/or have hyperlinks).</td>
</tr>
</tbody>
</table>

**Invariants**

*items_exists*: items /= void and then not items.is_empty

**FIGURE 7** illustrates the visual appearance of a typical **DV_PARAGRAPH**.

```
xxxxx xxx xxx xxxxxxx xx xx xxxxxxx xxx xxxxxxxxx xxx xxx
xxxxxxxxxxxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
xxxxxx a xxxxxxx xxx xx xxx xxx xxx xxx xxx xxx xxx
```

**FIGURE 7** PARAGRAPH visual structure
6 Quantity Package

6.1 Overview

The data_types.quantity package is illustrated in FIGURE 8. Dates and Times are found in the next section.

FIGURE 8 rm.data_types.quantity Package
6.1.1 Requirements

Ordinal Values
Medicine is one domain in which symbols representing relative magnitudes are commonly used, without exact values being known, or often, being knowable. The main purpose is usually to classify patients into groups for which different decisions might be made. Thus, while approximate ranges (technically speaking - “fuzzy intervals”) might be stated (such as for a urinalysis), concrete values are not of interest, only categories are. Take for example the characterisation of “pain” as being “mild”, “medium”, “severe”, or the reflex response to tendon percussion as “-”, “+/-”, “+”, “++”, “+++”, “++++”. There may be no way to scientifically quantify such values because they reflect a subjective experience of the patient or informal judgement by clinician.

Similarly, even though the symbolic values for haemolysed blood in a urinalysis have approximate ranges stated for them, as shown here, these values are not usable in any way.

- “neg”, “trace” (10 cells/µl)
- “small” (<25 cells/µl)
- “moderate” (<80 cells/µl)
- “large” (>200 cells/µl))

An argument for recording such values sometimes put forward is that comparisons might want to be made between the ranges quoted by two laboratories for the same symbol (e.g. “moderate”). There are a number of counter-arguments. Firstly, such comparisons are a poor attempt at “normalisation”, an activity which is the business of pathologists, not EHR users. Secondly, the symbolic values are often arrived at by the tester making a judgement of colour on a strip, which while an adequate (and cost-effective) approach for classifying, is not a valid means of quantifying a value. Lastly, in most cases, if a quantified point value or range is desired, or available, then it will be used - meaning that the appropriate quantitative data type can be used, rather than an the ordinal type.

Countable Things
An common kind of data value in medicine is the dimensionless countable quantity, e.g. “number of doses: 2”, “number of previous pregnancies: 1”, “number of tablets: 3”. This type can only ever be an integer number, but needs to be convertible to real numbers for statistical purposes, for example for a study of average number of pregnancies per couple.

Dimensioned Quantities
The most common kind of quantity is a measured, dimensioned quantity. Anything which is measurable (rather than countable) involves a number of data aspects, namely:

- a magnitude whose value is a real number;
- the physical property being measured, with the appropriate units;
- a concept of precision, i.e. to what number of decimal places the value is recorded;
- a concept of accuracy, i.e. the known or assumed error in the measurement due to instrumentation or human judgement.

Examples of dimensioned quantities include:

- systolic BP: 110 mmHg
- height: 178 cm
- rate of asthma attacks: 7 /week
- weight loss: 2.5 kg
Quantity Ratios
A not uncommon type in science and medicine is the ratio, which is used in situations like the following:

- 1:128 (a titer)
- 250 mg / 500 ml (solute/solvent)

Ratios in general have real number values, even if many actual examples appear to be integer ratios.

Ranges
Quantity ranges are ubiquitous in science and medicine, and may be used for any kind of quantity, from ordinals to measured quantities. Examples include:

- healthy weight range, e.g. 48kg - 60kg
- normal range for urinalysis in pregnancy - protein, e.g. “nil” - “trace”

Reference Ranges
Reference ranges are an important aspect of laboratory data, and may be relevant for any of the types described above. The general form of a reference range found in a pathology result indicates what is considered the “normal” range for a measured value. Examples of reference ranges:

- normal range for serum Na is 135 - 145 mmol/L.
- desirable total cholesterol: < 5.5 mmol/L (strictly this probably should be 2.0 - 5.5 mmol/L, but is not usually quoted this way as low cholesterol is not considered a problem.)

Ranges can also be quoted for drug administrations, in which case they are usually thought of as the “therapeutic” range. For example, the anticonvulsant drug Carbamazepine has a therapeutic range of 20 - 40 µMol/L. In some cases, there are multiple ranges associated with a drug, for example, Salicylate has a therapeutic range of 1.0 - 2.5 mmol/L and a toxic range > 3.6 mmol/L.

Various examples occur in which multiple ranges may be stated, including the following.

- The administration recomendations for drugs which depend on the particular patient state. For example, the therapeutic range of Cyclosporin (an immunosuppresant) is a function of time post-transplant for the affected organ, e.g. kidney: < 6 months: 250 - 350 µg/L, > 6 months: 100 - 200 µg/L.
- Normal ranges for blood IgG, IgA, IgM which vary significantly with the age in months from birth.
- Progesterone and pituitary hormones have ranges which are different for different phases of the menstrual cycle and for menopause. This may result in 4 or 5 ranges given for one result. Only one will apply to any particular patient - but the exact phase of the cycle may be unknown - so the ranges may need to be associated with the value with no 'normal' range.

Where there are multiple ranges, the important question is: which range information is relevant to the actual data being recorded for the patient? In theory, only the range corresponding to the particular patient situation should be used, i.e. the range which applies after taking into account sex, age, smoking status, “professional athlete”, organ transplanted, etc. In most cases, this is a single “normal” range, or a pair of ranges, typically “therapeutic” and “critical”. However, practical factors complicate things. Firstly, data is often supplied from pathology labs along with some or all of the applicable reference ranges, even though only some could possibly apply. This is particularly the case if the laboratory has no other data on the patient, and cannot evaluate which range applies. The requirement for faithfulness of recording might be extended to reference data supplied by laboratories, regardless of how irrelevant or arbitrarily chosen the reference data is, meaning that such data has to be stored in...
the record anyway. Secondly, there may be circumstances in which physicians want a number of reference ranges, even while knowing that only one range is applicable to the datum. Ranges above and below the relevant one might be useful to a physician wishing to determine how far out of range the datum is.

6.1.2 Design

Basic Model

In order to make sense of the requirements in a systematic way, a proper typology for quantities is needed. The most basic characteristic of all values typically called “quantities” is that they are ordered, meaning that the operator “<” (less-than) is defined between any two values in the domain. An ancestor class for all quantities called \texttt{DV\_ORDERED} is accordingly defined. This type is subtyped into ordinals and true quantities, represented by the classes \texttt{DV\_ORDINAL} and \texttt{DV\_QUANTIFIED} respectively. \texttt{DV\_ORDINAL} represents data values whose exact numeric values are not known, and which use symbolic renderings instead, such as “+”, “++”, “+++”, or “mild”, “medium”, “severe”. In contrast, instances of \texttt{DV\_QUANTIFIED} and all its subtypes have precise numeric magnitudes. \texttt{DV\_QUANTIFIED} itself introduces the concept of magnitude and accuracy, the latter of which is explained in more detail below. Its two subtypes - the abstract \texttt{DV\_MEASURABLE} - and the concrete \texttt{DV\_COUNT} reflect the two basic types of quantified value which exist. Measurable quantities are those which measure an amount of a physical quantity, while countable quantities are used to count entities in the real world.

The type \texttt{DV\_COUNT} has an integer magnitude and is used to record dimensionless naturally countable things such as number of previous pregnancies, number of steps taken by a recovering stroke victim and so on. There are no units or precision. Countable quantities can be used to create instances of \texttt{DV\_QUANTITY}, such as during a statistical study which average tobacco consumption over a time period. Such a computation might cause the creation of \texttt{DV\_QUANTITY} objects representing values like \{magnitude = 5.85, units = '/ week'\}

There are two subtypes of \texttt{DV\_MEASURABLE}. \texttt{DV\_QUANTITY} is used to represent amounts of measurable things, and has a real number magnitude, precision and units. The units attribute contains the scientific unit in a parsable form defined by the Unified Code for Units of Measure (UCUM) [8]. A valid units string always implies a measured property, such as “force” or “pressure”. The property of a Quantity can conveniently constrained in archetypes, e.g. to “pressure”, which would allow any pressure unit. Unit strings can be compared to determine if they measure the same property (e.g. “bar” and “Pa” are both units corresponding to the property “pressure”), which enables the is_strictly_comparable_to function defined on \texttt{DV\_ORDERED} to be properly specified on \texttt{DV\_MEASURABLE}.

It is important to note that while these semantics will allow comparison of e.g. two pressures recorded in mbar and mmHg, or even two accelerations whose units are “m.s^\text{-2}” and “m/s^2”, they provide no guarantee that this is a sensible thing to do in terms of clinical semantics: comparing a blood pressure to an atmospheric pressure for example may or may not make any sense. It is not within the scope of the quantity package to express such semantics: this is up to application software which uses Quantities found in specific places in the data.

Accuracy and Uncertainty

Theoretically, “accuracy” should not be included in a model for \texttt{QUANTIFIED} values, because it is an artifact of a measuring process and/or device, not of a quantity itself. For example, a weight of “82 kg +/-5%” can be represented in two parts. The “82kg” is represented as a \texttt{DV\_QUANTITY}, while the “+/-5%” may be included in the protocol description of the weighing instrument (recorded in an ENTRY),
since this is where the error comes from. However, for practical purposes, for any measured quantity for which accuracy was recorded, it is quite likely that the accuracy will be required in computations on the quantity, especially for statistical population queries in which measurement error must be disambiguated from true correlation. It is therefore included as an attribute of `DV_QUANTIFIED`. If not used, its value is 0.

The notion of “uncertainty” is understood as a subjective judgement made by the clinician, indicating that he/she is not certain of a particular statement. It is not the same as accuracy: uncertainty may apply to non-quantified values, such as subjective statements, and it is not an aspect of objective measurement processes, but of human confidence. Where the uncertainty is due to subjective memory e.g. “I think my grandfather was 56 when he died”, the uncertainty is simply recorded as another value, along with the main data item being recorded. Uncertainty is therefore not directly modelled in the openEHR data types, but appears instead in particular archetypes.

**Ranges**

Ranges are modelled by the generic type `DV_INTERVAL<T:DV_ORDERED>` which enables a range of any of the other quantity types (except ratio) to be constructed. This allows any subtype of `DV_ORDERED` to occur as a range as well.

**Quantity Ratios**

The `DV_QUANTITY_RATIO` type is provided for representing ratios, and consists simply of two `DV_QUANTIFIED` instances which may be of variable concrete type.

**Reference Ranges**

The approach taken in the model for modelling reference ranges is to provide the ability to add any number of named reference ranges to a `DV_QUANTIFIED` (i.e. date/time types and quantities), using the type `REFERENCE_RANGE`.

**Customary Quantities**

The subtype `DV_CUSTOMARY_QUANTITY` corresponds to quantities which are expressed in terms of an arbitrary arrangement of values and units. Examples of the latter include imperial measurements of weight and length (which are not modelled explicitly here, since they are either assumed to be recorded in metric, or else in the smallest of the relevant units, such as inches or ounces), and the time-related types in the “social/biological” time domain, which are recorded in year/month/day and/or hour/minute/second format. Instances of any customary type are convertible to instances of `DV_QUANTITY`. Time in finer time domains, e.g. chemical, atomic domains etc is recorded using `DV_QUANTITY`, with `units` set to “s” (seconds). [To be completely correct, this model would include a subclass `DV_CUSTOMARY_QUANTITY` for all time types, say `DV_TIME_QUANTITY`, whose `units` is set to “time”. However, it is not yet clear that there are any other customary types required in clinical medicine (e.g. imperial weights or lengths which need to be represented in parts rather than their smallest possible unit).]

**Statistical Reference Data**

To Be Continued:

### 6.2 Class Descriptions
### 6.2.1 DV_ORDERED Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_ORDERED (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract class defining the concept of ordered values, which includes ordinals as well as true quantities. It defines the functions ‘&lt;’ and is_strictly_comparable_to, the latter of which must evaluate to True for instances being compared with the ‘&lt;’ function, or used as limits in the DV_INTERVAL&lt;T&gt; class.</td>
</tr>
<tr>
<td>Use</td>
<td>Data value types which are to be used as limits in the DV_INTERVAL&lt;T&gt; class must inherit from this class, and implement the function is_strictly_comparable_to to ensure that instances compare meaningfully. For example, instances of DV_QUANTITY can only be compared if they measure the same kind of physical quantity.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_VALUE</td>
</tr>
</tbody>
</table>

#### Abstract

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>infix ‘&lt;’ (other: like Current): Boolean</td>
<td>Tests if this item is less than other, which must be of the same concrete type.</td>
</tr>
<tr>
<td>is_strictly_comparable_to (other: like Current): Boolean</td>
<td>Test if two instances are strictly comparable.</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference_ranges: List &lt;REFERENCE_RANGE&lt;like Current&gt;&gt;</td>
<td>optional tagged ranges for this value in its particular measurement context</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_normal: Boolean</td>
<td>Value is in the normal range if there is one, otherwise True.</td>
</tr>
<tr>
<td>normal_range: REFERENCE_RANGE&lt;like Current&gt;</td>
<td>If there are reference ranges, returns the reference range whose meaning is “normal”</td>
</tr>
<tr>
<td>is_simple: Boolean</td>
<td>True if this quantity has no reference ranges, or accuracy.</td>
</tr>
</tbody>
</table>
6.2.2 **DV_INTERVAL<T : DV_ORDERED>** Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_INTERVAL&lt;T : DV_ORDERED&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Generic class defining an interval (i.e. range) of a comparable type. An interval is a contiguous subrange of a comparable base type.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Used to define intervals of dates, times, quantities (whose units match) and so on. The type parameter, T, must be a descendant of the type <strong>DV_ORDERED</strong>, which is necessary (but not sufficient) for instances to be compared (<strong>strictly_comparable</strong> is also needed). Without the <strong>DV_INTERVAL</strong> class, quite a few more <strong>DV</strong> classes would be needed to express logical intervals, namely interval versions of all the date/time classes, and of quantity classes. Further, it allows the semantics of intervals to be stated in one place unequivocally, including the conditions for strict comparison. The basic semantics are derived from the class <strong>INTERVAL&lt;T&gt;</strong>, described in the support RM.</td>
</tr>
<tr>
<td><strong>ISO 18308</strong></td>
<td>STR 3.13</td>
</tr>
<tr>
<td><strong>CEN</strong></td>
<td>Time Interval; also includes a measurement range data type but not the ability to specify if minimum or maximum values are inclusive.</td>
</tr>
<tr>
<td><strong>Synapses</strong></td>
<td>QuantityRange + ability to specify if the range is inclusive or exclusive separately of the maximum and minimum values.</td>
</tr>
<tr>
<td><strong>GEHR</strong></td>
<td><strong>G1_QUANTITY_RANGE</strong></td>
</tr>
<tr>
<td><strong>HL7</strong></td>
<td><strong>IVL&lt;T:QTY&gt;</strong></td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td><strong>DATA_VALUE, INTERVAL&lt;T&gt;</strong></td>
</tr>
<tr>
<td><strong>Invariants</strong></td>
<td><strong>Limits_consistent</strong>: (not upper_unbounded and not lower_unbounded) implies (lower.is_strictly_comparable_to(upper) and lower &lt;= upper)</td>
</tr>
</tbody>
</table>
### 6.2.3 REFERENCE_RANGE<T:DV_ORDERED> Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>REFERENCE_RANGE&lt;T:DV_ORDERED&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Defines a named range to be associated with any ORDERED datum. Each such range is particular to the patient and context, e.g. sex, age, and any other factor which affects ranges.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>May be used to represent normal, therapeutic, dangerous, critical etc ranges.</td>
</tr>
<tr>
<td><strong>ISO 18308</strong></td>
<td>STR 3.13</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1..1</strong></td>
<td><strong>meaning</strong>: DV_TEXT</td>
</tr>
<tr>
<td><strong>1..1</strong></td>
<td><strong>range</strong>: DV_INTERVAL&lt;T&gt;</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>is_in_range</strong> (val: T): Boolean</td>
<td>Indicates if the value ‘val’ is inside the range</td>
</tr>
</tbody>
</table>

#### Invariants

- **Meaning_exists**: meaning /= Void
- **Range_exists**: range /= Void
- **Range_is_simple**: (range.lower_unbounded or else range.lower.is_simple) and (range.upper_unbounded or else range.upper.is_simple)

### 6.2.4 DV_ORDINAL Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_ORDINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Models rankings and scores, e.g. pain, Apgar values, etc, where there is a) implied ordering, b) no implication that the distance between each value is constant, and c) the total number of values is finite.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Used for recording any clinical datum which is customarily recorded using symbolic values. Example: the results on a urinalysis strip, e.g. {neg, trace, +, ++, +++} are used for leucocytes, protein, nitrites etc; for non-haemolysed blood {neg, trace, moderate}; for haemolysed blood {neg, trace, small, moderate, large}.</td>
</tr>
<tr>
<td><strong>ISO 18308</strong></td>
<td>STR 3.2</td>
</tr>
<tr>
<td><strong>HL7</strong></td>
<td>Quantity (QTY)</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>DV_ORDERED</td>
</tr>
</tbody>
</table>
The openEHR Data Types Information Model
Quantity Package
Rev 2.0

---

### CLASS

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: Integer</td>
<td>ordinal position in enumeration of values.</td>
</tr>
<tr>
<td>symbol: DV_CODED_TEXT</td>
<td>Coded textual representation of this value in the enumeration, which may be strings made from “+” symbols, or other enumerations of terms such as “mild”, “moderate”, “severe”, or even the same number series as the values, e.g. “1”, “2”, “3”.</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>limits: REFERENCE_RANGE &lt;DV_ORDINAL&gt;</td>
<td>limits of the ordinal enumeration, to allow comparison of an ordinal value to its limits.</td>
</tr>
<tr>
<td>infix ‘&lt;’ (other: like Current): Boolean ensure value &lt; other.value implies Result</td>
<td>True if types are the same and values compare</td>
</tr>
<tr>
<td>is_strictly_comparable_to (other: like Current): Boolean ensure symbol.is_comparable (other.symbol) implies Result</td>
<td>True if symbols come from same vocabulary, assuming the vocabulary is a subset or value range, e.g. “urine:protein”.</td>
</tr>
</tbody>
</table>

#### Invariants

- Value_valid: value > 0
- Symbol_exists: symbol /= Void
- Limits_valid: limits /= Void and then limits.meaning.is_equal(“limits”)
- Reference_range_valid: reference_ranges /= Void and then reference_ranges.has(limits)

---

### 6.2.5 DV_QUANTIFIED Class

#### CLASS

##### DV_QUANTIFIED (abstract)

**Purpose**

Abstract class defining the concept of true quantified values, i.e. values which are not only ordered, but which have a magnitude, and for which the addition and difference operations can be defined.

**OMG HDTF**

COAS::Measurement.

**Synapses**

Attributes in the Quantity class for unit and accuracy (double plus units)

**HL7**

Quantity (QTY)

**Inherit**

DV_ORDERED
**CLASS**

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_QUANTIFIED (abstract)</th>
</tr>
</thead>
</table>

### Abstract

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>magnitude</code>: Ordered_Numeric</td>
<td>Numeric value of the quantity in canonical (i.e. single value) form. Implemented as constant, function or attribute in subtypes as appropriate. The type Ordered_numeric is mapped to the available appropriate type in each implementation technology.</td>
</tr>
<tr>
<td><code>infix '+' (other: diff_type): like Current</code></td>
<td>Sum of this quantity and another whose formal type must be the difference type of this quantity.</td>
</tr>
<tr>
<td><code>infix '-' (other: diff_type): like Current</code></td>
<td>Difference of this quantity and another whose formal type must be the difference type of this quantity type.</td>
</tr>
<tr>
<td><code>diff_type</code>: DV_QUANTIFIED</td>
<td>Type of quantity which can be added or subtracted to this quantity. Usually the same type, but may be different as in the case of dates and times.</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>accuracy: Real</code></td>
<td>accuracy of measurement instrument or method which applies to this specific instance of DV_QUANTIFIED, expressed either as a half-range percent value (accuracy_is_percent = True) or a half-range quantity. A value of 0 means that accuracy was not recorded.</td>
</tr>
<tr>
<td><code>accuracy_is_percent: Boolean</code></td>
<td>If True, indicates that when this object was created, accuracy was recorded as a percent value; if False, as an absolute quantity value.</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is_valid_percentage (x: Numeric): Boolean</code></td>
<td>Test whether a number is a valid percentage</td>
</tr>
</tbody>
</table>

### Invariants

- **Magnitude_exists**: `magnitude /= Void`
- **Accuracy_validity**: `accuracy_is_percent implies is_valid_percentage(accuracy)`
### 6.2.6 DV_MEASURABLE Class

**CLASS**  
**DV_MEASURABLE (abstract)**

**Purpose**
Abstract class defining the concept of true quantified values, i.e. values which are not only ordered, but whose magnitude is meaningful as well.

Units were inspired by the Unified Code for Units of Measure (UCUM), developed by Gunther Schadow and Clement J. McDonald of The Regenstrief Institute [8].

**CEN**
[unit] exists as an attribute of the measurement data value class.

**OMG HDTF**
COAS::Measurement.

**Synapses**
Attributes in the Quantity class for unit and accuracy (double plus units)

**HL7**
Quantity (QTY)

**Inherit**
DV_QUANTIFIED

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>units</td>
<td>String</td>
<td>Stringified units, expressed in UCUM unit syntax, e.g. &quot;kg/m2&quot;, &quot;mm[Hg]&quot;, &quot;ms-1&quot;, &quot;km/h&quot;. Implemented accordingly in sub-types.</td>
</tr>
</tbody>
</table>

**Invariants**
Units_valid: units /= void

### 6.2.7 DV_QUANTITY Class

**CLASS**
**DV_QUANTITY**

**Purpose**
Quantitified type representing “scientific” quantities, i.e. quantities expressed as a single value and optional units.

**Use**
Can also be used for time durations, where it is more convenient to treat these as simply a number of seconds rather than days, months, years.

**ISO 18308**
STR 3.2 - 3.4

**CEN**
Quantifiable Data Item; Measurement data value class.

**OMG HDTF**
COAS::Numeric.

**Synapses**
Quantity

**GEHR**
G1_QUANTITY

**HL7**
PhysicalQuantity (PQ)
### 6.2.8 DV_COUNT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Countable quantities.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for countable types such as pregnancies and steps (taken by a physiotherapy patient), number of cigarettes smoked in a day.</td>
</tr>
<tr>
<td>Misuse</td>
<td>Not used for amounts of physical entities (which all have units)</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.2 - 3.4</td>
</tr>
<tr>
<td>HL7</td>
<td>INT</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_QUANTIFIED</td>
</tr>
</tbody>
</table>
6.2.9 Units Syntax

The BNF syntax specification of the units string, adapted from [8] is as follows:

**Parse Specification**

```
units ::= '/' exp_units
| units '.' exp_units
| units '/' exp_units
| exp_units
```

```
exp_units ::= unit_group exponent | unit_group
```

```
unit_group ::= PREFIX annot_unit
| annot_unit
| '(' exp_units ')' | factor
```

```
annot_unit ::= unit_name
| unit_name '{' ANNOTATION '}'
| '{' ANNOTATION '}'
```

```
factor ::= Integer
```

```
exponent ::= SIGN Integer | Integer
```

**Lexical Specification**

```
PREFIX ::= 'Y' | 'Z' | 'E' | 'P' | 'T' | 'G' | 'M' | 'k' | 'h' | 'da'
| 'd' | 'c' | 'm' | 'µ' | 'n' | 'p' | 'f' | 'a' | 'z' | 'y'
```

```
UNIT_NAME ::= [a-zA-Z%_]+ ; from unit tables
```

```
ANNOTATION ::= [a-zA-Z'.]+ ; from unit tables
```

```
SUFFIX ::= [a-zA-Z0-9'\_]+ ; from unit tables
```

```
SIGN ::= '+' | '-'
```

```
Integer ::= [0-9]+^
```

This proposal is comprehensive, covering all useful unit systems, including SI, various imperial, customary measures, and some obscure measures, as well as clinically specific additions. Metric prefixes, meaning-changing textual suffixes (e.g. “[Hg]” in “mm[Hg]”) and non-meaning-changing annotations (e.g. “kg {total}”) are recognised. With this syntax, units can be simply expressed in strings such as:

“kg/m^2”, “m.s^-1”, “km/h”, “mm[Hg]”
and so on.

### 6.2.10 DV_QUANTITY_RATIO Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_QUANTITY_RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Models a ratio of quantities.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for recording specified administration dosages (e.g. 5 mg / 100 ml), drug amounts based on body weight (e.g. 1 tablet / 10 kg), and titers (e.g. 1:128). Note that the units representation in single QUANTITYs caters for any ratio in which the units are expressible in unitary (i.e. denominator = 1) form. Thus, a QUANTITY_RATIO of “2 g / 250ml” could be expressed as a QUANTITY of “8 g/l”.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Should not be used to represent things like blood pressure which are often written using a ‘/’ character, giving the misleading impression that the item is a ratio, when in fact it is a structured value. E.g. visual acuity “6/24” is not a ratio.</td>
</tr>
</tbody>
</table>

**ISO 18308**

STR 3.6

**OMG HDTF**

COAS::Ratio.

**Synapses**

Numeric class

**GEHR**

G1_QUANTITY_RATIO

**HL7**

Ratio (RTO). In HL7, the RTO type is used only for ratios of reals or integers, and does not seem to allow for ratios of dimensioned quantities.

**Inherit**

DATA_VALUE

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>numerator: DV_QUANTIFIED</td>
<td>numerator of ratio</td>
</tr>
<tr>
<td>1..1</td>
<td>denominator: DV_QUANTIFIED</td>
<td>denominator of ratio</td>
</tr>
</tbody>
</table>

**Invariants**

Numerator_exists: numerator /= Void  
Denominator_exists: denominator /= Void

### 6.2.11 DV_CUSTOMARY_QUANTITY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_CUSTOMARY_QUANTITY (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent class of quantity types which are expressed in a form other than the standard scientific, i.e. one value, one unit form</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_QUANTIFIED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>
### Class: DV_CUSTOMARY_QUANTITY (abstract)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>units: String</td>
<td>Constant</td>
</tr>
</tbody>
</table>

#### Invariants

- Units_valid: units.is_equal("s")

### Function to convert a customary quantity to a scientific one for comparison or other purposes.

- **to_quantity**: DV_QUANTITY
- **result_exists**: Result /= void
7 Date Time Package

7.1 Overview

The data_types.quantity.date_time package includes three absolute date/time concepts: DV_DATE, DV_TIME, DV_DATE_TIME, a relative concept: DV_DURATION, and the concept of partial dates and times, via DV_PARTIAL_DATE, DV_PARTIAL_TIME. The date_time package is illustrated in FIGURE 9.

FIGURE 9 rm.data_types.quantity.date_time Package

7.1.1 Requirements

Standard Date/Time

The basic requirement is for types which represent the following concepts:

- Date: a type which records year, month and day in month. Examples include date of birth, date of onset of a problem
- Time: a type which records hour, minute, second, and timezone. Examples include time of meal, time of day when a problem recurs. Timezone is required in a shared EHR repository so that times of clinical events which occurred in different timezones are comparable; this includes specialised pathology tests which might be done in another country.
- **Date_time**: a type which records year, month, day, hour, minute, second, and timezone. Examples include date & time of death, timestamp of any observation. Timezone required for the same reason as in *Time*.

- **Duration**: a type which records duration of an event or (in)activity, as days, hours, minutes, and seconds.

**Fuzzy or Incomplete Date/Times**

Partial or uncertain date/times have to be catered for in clinical medicine. It is common for patients to be unsure about dates and durations. Requirements for partial date/times include the following.

- In practical clinical experience, it turns out that partial dates represent the majority of incomplete date/times. For dates, one of the following rules applies to any instance:
  - only the year is known
  - only the year and month are known

  If not even the year is known, then the date is obviously extremely approximate and it would probably be unsafe to represent it computationally. However, if computatable representation was needed in this case, a date interval can be used. A pedantic example which breaks these rules is someone who claims to be born on “a Monday at the start of May in 1934” (i.e. day but not date unknown). Either the clinician determines what date the first Monday in May 1934 actually was and record that (assuming the patient’s way of accurately remembering just happens to be via day rather than date), or else records a partial date of the form “May 1934” (in ISO 8601 form, “1934-05”) if they determine that the patient really is unsure.

- Sometimes incomplete times are recorded, which follow the same rule that either the hours or both the hours and minutes are present. Examples:
  - recordings by instruments which only generate hh:mm values (i.e. no seconds);
  - recordings by patients who report approximate times of events;
  - recordings by clinicians who use approximate times in administrations, e.g. “take insulin at 8am” really means something like 8am +/- 30 mins.

- Imprecise durations such as “2 - 3 hrs” need to be recordable in a computable form.

To satisfy the faithfulness requirement for health record recording it should always be possible to record the narrative form of the datum provided by the patient as well as the formal form.

**7.1.2 Design**

Date/time values are somewhat special in the realm of data types. They can be expressed in standard Quantity form, i.e. as a number of seconds, or they can be expressed in their “customary” form, in which the standard structure of {value, unit} and metric relationships between orders of magnitude do not hold. The customary form is what we are used to using with date/time quantities which relate to affairs in the social time domain, such as births, deaths, ages, and times and durations of events which we remember; in these cases it is expressed using the familiar year/month/date/hour/minute/second system, in which the relationships between each successive unit of time is non-metric. Scientific observations, mostly at finer granularities of time, are usually expressed using the standard Quantity form rather than the customary form.

In clinical medicine, both types are used ubiquitously, and there is a need to be able to process date/time quantities both in their customary form, and in their scientific form. Consequently, the openEHR model takes the approach that date/time quantities are a subtype of the class *DV_CUSTOMARY_QUANTITY*, itself a kind of *DV_QUANTIFIED*. Each subtype can easily be converted
to the scientific DV_QUANTITY form, while DV_QUANTITYs can be converted to customary form if an appropriate one exists.

All date/time types in this specification fall into two broad groups: absolute and relative. Absolute date and time classes all inherit from the abstract class DV_WORLD_TIME, i.e. values measuring time in a geographically located real world context. Such date/times measure absolute time: thus real world dates measure calendrical time, while real world times measure clock time from midnight. Consequently, all must include timezone information, ensuring that all instances of such dates and times are situated on the same timeline, notionally that which is experienced at longitude 0°.

The relative category contains only the concept DV_DURATION, which expresses elapsed time from some arbitrary time point. DV_DURATION is used for expressing durations of clinical phenomena, differences between two world times, and the size of the timezone offset of any world time.

Fuzzy and Incomplete Date/Times

Clearly, quite complex models for fuzzy date/times and uncertainty are possible. The approach used here takes into account the known needs for representing partially known date/time data, while balancing that with the need to avoid incomprehensibly complex fuzzy types whose generality would really only apply to a tiny percent of difficult cases. Thus, the basis for modelling incomplete date/times is as follows.

- The modelling problem relates only to date/time quantities that need to be computable. For extremely imprecise date/times, if the clinician feels the need, s/he can record it as narrative text.
- A function is provided giving applications easy access to a DV_INTERVAL<DV_DATE> enclosing the maximum range which could be implied by the originally stated date. Similarly for partial times.
- For imprecise durations, an interval should be used, i.e. DV_INTERVAL<DV_DURATION>. In this way durations like “2 - 3 hrs” can be represented, and still be computable.

Based on the above considerations, the requirements for partial types are satisfied by the classes DV_PARTIAL_DATE, DV_PARTIAL_TIME and DV_PARTIAL_DATE_TIME.

Calendars

A comment on calendars is in order. In this specification, the Gregorian calendar is assumed in all date types. Initially this may seem like a culturally insensitive approach, but in fact it makes sense in computational terms, for both users of the Gregorian calendar and any other calendars, e.g. Julian, Islamic, Baha’i, etc.

Arguments against allowing dates and date/times to be from any calendar include the following:

- Almost all dates on computer systems, including in regions such as the Indian sub-continent and the middle east, where alternate calendars are in use, are in the Gregorian system. This is likely to be the case for some time, and may always be the case, regardless of the continued use of other calendars for religious or other purposes (outside of health);
- If a calendar indicator were used in date quantities, all software, to be correct, would have to check the value to verify that it is in the expected calendar system, and to do something special if it is not - an added cost which is a possible source of bugs and which would rarely be used. The reality is that most software produced in the western world, India etc (possibly excepting open source software) would automatically assume the Gregorian calendar, and would be in error if ever it did receive EHR data containing dates from alternate calendars.
If/when other calendars are used in EHR or related systems, the users of those calendars will be aware of it, and include the appropriate conversion logic between Gregorian dates and their own, limiting the extra software work and quality issues to those users who actually need alternate calendars. If EHRs from such places are sent to a health care facility where Gregorian is the default, nothing special is needed to ensure that those records will contain dates comprehensible to the receiver.

The strategy is effectively that users requiring non-Gregorian dates in EHR and other health systems should include their own conversion code to and from Gregorian dates. This is no different from the same requirement for any particular group of users to include special software relating to particular kinds of user interfaces, language processing or other local/regional/cultural differences.

Algorithms for conversion between the Egyptian, Armenian, Khwarizmian, Persian, Ethiopian, Coptic, Republican, Macedonian, Syrian, Julian Roman, Gregorian, Islamic A, Islamic B, Baha’i and Saka calendars are described by Richards [7] and are based on the work of D. A. Hatcher (1986).

**Representation**

The classes described here define computational semantics, and do not in themselves define representation in particular implementation technologies. When a sharable, external representation has to be defined, such as in XML-schema or any other data format, ISO 8601 format is used. This is implied by the use of this standard for the `as_string` attribute of all date/time types here.

### 7.2 Definitions

The following symbolic definitions are used in the classes below:

- **Seconds_in_minute**: Integer = 60
- **Minutes_in_hour**: Integer = 60
- **Hours_in_day**: Integer = 24
- **Days_in_year**: Integer = 365
- **Days_in_week**: Integer = 7
- **Months_in_year**: Integer = 12
- **Middle_second_in_minute**: Integer = 30
- **Middle_minute_in_hour**: Integer = 30
- **Middle_day_in_month**: Integer = 15
- **Middle_month_in_year**: Integer = 6
- **Last_day_of_middle_month**: Integer is 30
- **Min_timezone**: DV_DURATION; Min_timezone.as_string = “-14:00”
- **Max_timezone**: DV_DURATION; Max_timezone.as_string = “+10:00”

Note that the timezone limits are set by where the international dateline is. Thus, time in New Zealand is quoted using +13:00, not -11:00.
7.3 Class Descriptions

7.3.1 DV_WORLD_TIME Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_WORLD_TIME (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract concept of time on the real world timeline. All dates assumed to be in the Gregorian calendar.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for recording dates and/or times in real world time.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_CUSTOMARY_QUANTITY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>timezone: DV_DURATION</td>
<td>offset from Universal Coordinated Time, in the range -1200 - +1200 (note that this can affect the date even if no time is recorded).</td>
<td></td>
</tr>
</tbody>
</table>

| Invariants | timezone_valid: timezone /= Void and then (timezone >= Min_timezone and timezone <= Max_timezone) |

7.3.2 DV_DATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents an absolute point in time, as measured on the Gregorian calendar, and specified only to the day.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for recording dates in real world time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ISO 18308</th>
<th>STR 3.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEN</td>
<td>TOCD choice Quantifiable Observation Data Item</td>
</tr>
<tr>
<td>Synapses</td>
<td>DTValue attribute of DateTime class (this does not distinguish the representation of dates and of times)</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_DATE</td>
</tr>
<tr>
<td>HL7</td>
<td>PointInTime (TS). Note that this type simply measures a number of seconds since an epoch, with a timezone. These values are convertible to y/m/d form via the calendar attribute of TS.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_CUSTOMARY_QUANTITY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff_type: DV_DURATION</td>
<td>Difference type for DV_DATE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>
The openEHR Data Types Information Model

Date Time Package
Rev 2.0

7.3.3 DV_TIME Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>year: Integer</td>
</tr>
<tr>
<td></td>
<td>year</td>
</tr>
<tr>
<td>1..1</td>
<td>month: Integer</td>
</tr>
<tr>
<td></td>
<td>month in year</td>
</tr>
<tr>
<td>1..1</td>
<td>day: Integer</td>
</tr>
<tr>
<td></td>
<td>day in month</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_valid_date</td>
<td>y, m, d:Integer</td>
<td>date is valid in Gregorian calendar</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-- complex correctness condition</td>
</tr>
<tr>
<td>magnitude</td>
<td>Integer</td>
<td>numeric value of the date as seconds since the calendar origin point 1/1/0001</td>
</tr>
<tr>
<td>to_quantity</td>
<td>DV_QUANTITY</td>
<td>Convert to a number of days (the unit “d” is an ISO1000 unit).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result.units.is_equal(“d”)</td>
</tr>
<tr>
<td>as_string</td>
<td>String</td>
<td>Result = “yyyy-MM-dd”, following ISO 8601 (see <a href="http://www.cl.cam.ac.uk/~mgk25/iso-time.html">http://www.cl.cam.ac.uk/~mgk25/iso-time.html</a>)</td>
</tr>
</tbody>
</table>

### Invariants

**Validity**: is_valid_date(year, month, day)

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents an absolute point in time from an origin usually interpreted as meaning the start of the current day, specified to the second.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for recording real world times, rather than scientifically measured fine amounts of time.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.7, 3.10</td>
</tr>
<tr>
<td>CEN</td>
<td>TOCD choice Quantifiable Observation Data Item</td>
</tr>
<tr>
<td>Synapses</td>
<td>DTVvalue attribute of DateTime class (this does not distinguish the representation of dates and of times)</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_TIME</td>
</tr>
<tr>
<td>HL7</td>
<td>PointInTime (TS). Note that this type simply measures a number of seconds since an epoch. These values are convertible to ymd form via the calendar attribute of TS.</td>
</tr>
</tbody>
</table>
### 7.3.4 DV_DATE_TIME Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_DATE_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents an absolute point in time, specified to the second. In canonical form,</td>
</tr>
</tbody>
</table>

#### Inherit

- DV_WORLD_TIME

#### Abstract

- **diff_type**: DV_DURATION
  - Difference type for DV_TIME

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 hour</td>
<td>hour</td>
</tr>
<tr>
<td>1..1 minute</td>
<td>minute in hour</td>
</tr>
<tr>
<td>1..1 second</td>
<td>second in minute</td>
</tr>
<tr>
<td>0..1 fractional_second</td>
<td>fractional seconds</td>
</tr>
</tbody>
</table>

#### Functions

- **is_valid_time (h, m, s:Integer)**
  - Boolean
  - Result implies
  - $0 \leq h < \text{Hours\_in\_day}$
  - $0 \leq m < \text{Minutes\_in\_hour}$
  - $0 \leq s < \text{Seconds\_in\_minute}$
  - time is valid within 24h/60min/60sec system of time

- **to_quantity**: DV_QUANTITY
  - Result.units.is_equal("s")
  - Convert to a number of seconds (the unit “s” is a base SI unit).

- **magnitude**: Double
  - numeric value of the time as seconds since the start of day

- **as_string**: String
  - Result = “Thh:mm:ssZ”, following ISO 8601 (see [http://www.cl.cam.ac.uk/~mgk25/iso-time.html](http://www.cl.cam.ac.uk/~mgk25/iso-time.html)). ‘Z’ stands for either the literal ‘Z’ meaning “zero median”, i.e. Greenwich, or else a duration of the form “+/-hhmm”, e.g. “+1000”.

#### Invariants

- **Validity**: is_valid_time(hour, minute, second)
- **Fractional_second_valid**: fractional_second $\geq 0$ and fractional_second $< 1$
## Use

Used for recording a precise point in real world time.

## ISO 18308

STR 3.7, 3.10

## CEN

TOCD choice Quantifiable Observation Data Item

## OMG HDTF

COAS::DateTime

## Synapses

DTValue attribute of DateTime class (this does not distinguish the representation of dates and of times)

## GEHR

G1_DATE_TIME

## HL7

PointInTime (TS). Note that this type simply measures a number of seconds since an epoch, with a timezone. These values are convertible to y/m/d form via the calendar attribute of TS.

## Inherit

DV_WORLD_TIME

### Abstract

#### Signature

<table>
<thead>
<tr>
<th>diff_type: DV_DURATION</th>
</tr>
</thead>
</table>

#### Meaning

Difference type for DV_DATE_TIME

### Attributes

#### Signature

<table>
<thead>
<tr>
<th>1..1</th>
<th>year: Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>month: Integer</td>
</tr>
<tr>
<td>1..1</td>
<td>day: Integer</td>
</tr>
<tr>
<td>1..1</td>
<td>hour: Integer</td>
</tr>
<tr>
<td>1..1</td>
<td>minute: Integer</td>
</tr>
<tr>
<td>1..1</td>
<td>second: Integer</td>
</tr>
<tr>
<td>0..1</td>
<td>fractional_second: Double</td>
</tr>
</tbody>
</table>

#### Meaning

year

month in year

day in month

hour in day

minute in hour

second in minute

fractional seconds

### Functions

#### Signature

| is_valid_date_time(y, m, d, h, mi, s:Integer): Boolean |
| magnitude: Double |
| to_quantity: DV_QUANTITY ensure Result.units.is_equal("d") |

#### Meaning

date/time is valid within Gregorian calendar and within 24h/60min/60sec system of time

numeric value of the date/time as days since the calendar origin point

Convert to a number of days (the unit “d” is an ISO1000 unit).
### 7.3.5 DV_DURATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_DURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents a period of time with respect to a notional point in time, which is not specified. A sign may be used to indicate the duration is “backwards” in time rather than forwards.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for recording the duration of something in the real world, particularly when there is a need a) to represent the duration in customary format, i.e. days, hours, minutes etc, and b) if it will be used in computational operations with date/time quantities, i.e. additions, subtractions etc.</td>
</tr>
<tr>
<td>MisUse</td>
<td>Durations cannot be used to represent points in time, or intervals of time.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.10</td>
</tr>
<tr>
<td>CEN</td>
<td>Time Interval or Date Range or text description (pt 4)</td>
</tr>
<tr>
<td>GEHR</td>
<td>G1_DATE_TIME_DURATION</td>
</tr>
<tr>
<td>HL7</td>
<td>Interval of Point in Time, IVL&lt;TS&gt;. The width attribute provides the duration. IVL&lt;TS&gt; thus models an anchored duration.</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_CUSTOMARY_QUANTITY</td>
</tr>
</tbody>
</table>

#### Abstract

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff_type: DV_DURATION</td>
<td>Difference type for DV_DURATION</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>days: Integer</td>
<td>number of 24 hour days</td>
</tr>
<tr>
<td>hours: Integer</td>
<td>number of 60 minute hours</td>
</tr>
<tr>
<td>minutes: Integer</td>
<td>number of 60 second minutes</td>
</tr>
</tbody>
</table>
### 7.3.6 DV_PARTIAL_DATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_PARTIAL_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Represents a partially known date. All partial dates have an unknown day, by definition, else they would be represented as normal dates. The month_known flag indicates whether the month is also unknown.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for approximate birth dates, dates of death, etc.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.8</td>
</tr>
<tr>
<td>Synapses</td>
<td>FromDate attribute of DateTime class (this does not distinguish the representation of dates and of times)</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_DATE</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>month_known: Boolean</td>
<td>Indicates whether month in year is known. If so, the date is of the form “yyyy-mm”, if not, it is of the form “yyyy”</td>
</tr>
</tbody>
</table>
### 7.3.7 DV_PARTIAL_TIME Class

**CLASS** | **DV_PARTIAL_TIME**
--- | ---
**Purpose** | Represents a partially known time. All partial time have an unknown second, by definition, else they would be represented as normal times. The minute_known flag indicates whether the minute is also unknown.

**Use** | Used for approximate times of events and substance administrations.

**ISO 18308** | STR 3.8

**Synapses** | FromTime attribute of DateTime class (this does not distinguish the representation of dates and of times)

**Inherit** | DV_TIME

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 minute_known: Boolean</td>
<td>Indicates whether minute is known. If so, the time is of the form “hh:mm”, if not, it is of the form “hh”</td>
</tr>
</tbody>
</table>

**CLASS** | **DV_PARTIAL_DATE**
--- | ---
**Functions** | ** Meaning |
| enclosing_interval: DV_INTERVAL<DV_DATE> | Enclosing date range implied by this partial date. |
| magnitude: Integer ensure Result = enclosing_interval.mid-point.magnitude | canonical value of midpoint |
| as_string: String ensure Result follows ISO 8601 | |

**Invariants**

Enclosing_interval: month_known implies enclosing_interval.lower.day = 1 and enclosing_interval.upper.day = days_in_month(month, year) and not month_known implies enclosing_interval.lower.month = 1 and enclosing_interval.upper.month = Months_in_year and enclosing_interval.lower.day = 1 and enclosing_interval.upper.day = days_in_month(Months_in_year, year)
## 7.3.8 DV_PARTIAL_DATE_TIME Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_PARTIAL_DATE_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Represents a partially known date/time. All partial date/times have a known year, and an unknown second, by definition, else they would be represented as normal dates. The <code>month_known</code>, <code>day_known</code>, <code>hour_known</code> and <code>minute_known</code> flags indicates which of the other parts are unknown.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Used for approximate time stamps, e.g. the origin of a HISTORY in an OBSERVATION which is only partially known, etc.</td>
</tr>
<tr>
<td><strong>ISO 18308</strong></td>
<td>STR 3.8</td>
</tr>
<tr>
<td><strong>Synapses</strong></td>
<td><code>FromDate</code> attribute of DateTime class (this does not distinguish the representation of dates and of times)</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>DV_DATE_TIME</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td><code>month_known</code>: Boolean</td>
<td>Indicates whether month in year is known.</td>
</tr>
<tr>
<td>1..1</td>
<td><code>day_known</code>: Boolean</td>
<td>Indicates whether day in month is known.</td>
</tr>
<tr>
<td>1..1</td>
<td><code>hour_known</code>: Boolean</td>
<td>Indicates whether hour in day is known.</td>
</tr>
</tbody>
</table>
### CLASS

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_PARTIAL_DATE_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>minute_known: Boolean</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>enclosing_interval:</td>
<td>DV_INTERVAL&lt;DV_DATE_TIME&gt;</td>
<td>Enclosing date range implied by this partial date/time.</td>
</tr>
<tr>
<td>magnitude:</td>
<td>Double ensure</td>
<td>canonical value of midpoint</td>
</tr>
<tr>
<td>as_string:</td>
<td>String ensure</td>
<td>Result = follows ISO 8601</td>
</tr>
</tbody>
</table>

### Invariants

<table>
<thead>
<tr>
<th>Invariants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Month_known_validity:</td>
<td>not month_known implies not day_known</td>
</tr>
<tr>
<td>Day_known_validity:</td>
<td>not day_known implies not hour_known</td>
</tr>
<tr>
<td>Hour_known_validity:</td>
<td>not hour_known implies not minute_known</td>
</tr>
<tr>
<td>Enclosing_interval:</td>
<td>month_known implies enclosing_interval.lower.day = 1 and enclosing_interval.upper.day = days_in_month(month, year) and not month_known implies enclosing_interval.lower.month = 1 and enclosing_interval.upper.month = Months_in_year and enclosing_interval.lower.day = 1 and enclosing_interval.upper.day = days_in_month(Months_in_year, year) and minute_known implies enclosing_interval.lower.second = 1 and enclosing_interval.upper.second = Seconds_in_minute and not minute_known implies enclosing_interval.lower.minute = 1 and enclosing_interval.upper.minute = Minutes_in_hour and enclosing_interval.lower.second = 1 and enclosing_interval.upper.second = Seconds_in_minute</td>
</tr>
</tbody>
</table>
8 Time_specification Package

8.1 Overview

Time specification is about potentiality rather than actuality, and it needs its own types. The openEHR data_types.uri package provides such types, and is illustrated in FIGURE 10.

8.1.1 Requirements

One of the difficulties with time is expressing future times, since potential occurrences, durations, repetitions cannot be expressed in the same way as actual time. Complicating the problem is the fact that humans tend to use very customary (i.e. calendar-anchored) ways of specifying time, such as “every second Tuesday”, or “the first Sunday of the month”. In clinical medicine, future time is most commonly used to express when medications or other therapies are intended to take place. They are often anchored to the calendar, and can easily include repetitions.

As with other time types, there are both simple and complex cases to consider. One of the most common examples of time in the future is the timing for drug administrations, e.g. “once every four hours”. This could be represented as a simple periodic specification, consisting of a start point in time, a period, and a number of repetitions. The specification for taking blood sugar levels during a glucose test could be represented as a simple aperiodic series, e.g. “.5hr, 1hr, 2hr”. However, even common specifications for prescriptions e.g. “three times a day for seven days” start to become quite complex, for example, because “three times a day” might not mean literally 8 hours apart.

Some of the factors to consider in timing specifications are:

- period of repetition
- duration of activity being specified
- possible alignment to the calendar, e.g. “every 5th of the month”
• possible alignment to real world events e.g. “after meals”
• fuzziness

Because time is inherently “messy” (months do not all have the same number of days, leap years change the number of days in some years etc), and because the relationship we have with time can also be arbitrary (e.g. anchored to mealtimes etc), specifying linguistically obvious specifications formally is quite challenging.

8.1.2 Design
The HL7 version 3 data types for time specification appear to allow for all of the required possibilities. The syntax is based on the ISO 8601 standard [9]. It provides types which express:

• Periodic intervals (HL7v3 - PIVL<T:TS>) - allows period, duration, and calendar linking to be specified.
• Event-linked periodic intervals (HL7v3 - EIVL<T:TS>) - allows PIVLs to be linked to real-world events like meals.
• General timing specification (HL7v3 - GTS) - allows any time specification to be expressed, using a syntax which is equivalent to a series of IVL<TS> (i.e. intervals of DATE_TIME).

The HL7 syntax for time specification is encapsulated in equivalent openEHR types described here.

8.2 Class Descriptions

8.2.1 DV_TIME_SPECIFICATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_TIME_SPECIFICATION (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>This is an abstract class of which all timing specifications are specialisations. Specifies points in time, possibly linked to the calendar, or a real world repeating event, such as “breakfast”.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.9</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_VALUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>value: DV_PARSABLE</td>
<td>the specification, in the HL7v3 syntax for PIVL or EIVL types. See below.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>calendar_alignment: String</td>
<td>Indicates what prototypical point in the calendar the specification is aligned to, e.g. “5th of the month”. Empty if not aligned. Extracted from the ‘value’ attribute.</td>
<td></td>
</tr>
<tr>
<td>event_alignment: String</td>
<td>Indicates what real-world event the specification is aligned to if any. Extracted from the ‘value’ attribute.</td>
<td></td>
</tr>
</tbody>
</table>
8.2.2 DV.PERIODIC_TIME_SPECIFICATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV.PERIODIC_TIME_SPECIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Specifies periodic points in time, linked to the calendar (phase-linked), or a real world repeating event, such as “breakfast” (event-linked). Based on the HL7v3 data types PIVL&lt;T&gt; and EIVL&lt;T&gt;.</td>
</tr>
<tr>
<td>Use</td>
<td>Used in therapeutic prescriptions, expressed as INSTRUCTIONs in the openEHR model.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.9</td>
</tr>
<tr>
<td>CEN</td>
<td>The Duration data value class provides for the specification of time intervals, and also for a simple string description of the periodicity.</td>
</tr>
<tr>
<td>HL7</td>
<td>PIVL&lt;T&gt;, EIVL&lt;T&gt;</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_TIME_SPECIFICATION</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>period: DV_DURATION</td>
<td>The period of the repetition, computationally derived from the syntax representation. Extracted from the ‘value’ attribute.</td>
</tr>
<tr>
<td>ensure Result /= Void</td>
<td></td>
</tr>
<tr>
<td>calendar_alignment: String</td>
<td>Calendar alignment extracted from value.</td>
</tr>
<tr>
<td>event_alignment: String</td>
<td>Event alignment extracted from value.</td>
</tr>
<tr>
<td>institution_specified: Boolean</td>
<td>Extracted from value.</td>
</tr>
</tbody>
</table>

### Invariant

Value_valid: value.formalism.is_equal("HL7:PIVL") or value.formalism.is_equal("HL7:EIVL")

8.2.2.1 Phase-linked Time Specification Syntax

The syntactic form of phase-linked periodic time specifications (derived from the PIVL<T> spec HL7v3 ballot) is as follows.

“[“ interval “]“ /“ (“ difference “)“ [ “∅” alignment ] [ “IST” ]
Examples include:
- \([200004181100;200004181110]/(7d)@DW = \text{every Tuesday from 11:00 to 11:10 AM.}\)
- \([200004181100;200004181110]/(1mo)@DM" = \text{every 18th of the month 11:00 to 11:10 AM.}\)

A parse specification is as follows:

```
phase_linked_time_spec: pure_phase_linked_time_spec | pure_phase_linked_time_spec “IST”
pure_phase_linked_time_spec: phase |
  phase “@” alignment
phase: interval “/” “{” difference “}”
alignment: “DW” | etc /* terms from “HL7::CalendarCycle” domain */
difference: /* ISO 8601 for time difference */
interval: “[” interval_spec “]”
interval_spec: “;” |
  “;” date_time |
  date_time “;” date_time |
  date_time “;”

date_time: /* ISO 8601 for date/time string yyyy-mm-dd[hh:mm:ss] */
```

### 8.2.2.2 Event-linked Periodic Time Specification Syntax

Examples of event-linked periodic time specifications include:
- "PC+[1h;1h]" = \text{one hour after meal}
- "HS-[50min;1h]" = \text{one hour before bedtime for 10 minutes}

The following parse specification defines the syntax for event-related periodic time specifications.

```
event_linked_time_spec: event |
  event offset
event: “AC” | “ACD” | etc /* HL7 domain “HL7::TimingEvent” */
offset: “+” dur_interval |
  “-” dur_interval
dur_interval: /* ISO 8601 for duration interval */
```

### 8.2.3 DV_GENERAL_TIME_SPECIFICATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_GENERAL_TIME_SPECIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Specifies points in time in a general syntax. Based on the HL7v3 GTS data type.</td>
</tr>
<tr>
<td>Use</td>
<td>STR 3.9</td>
</tr>
</tbody>
</table>

ISO 18308
The class is the same structurally as the `DV_TIME_SPECIFICATION` parent. The syntax is the HL7 GTS syntax, defined by the following parse specification:

```
  general_time_spec: symbol | union | exclusion
    union: intersection ";" union | intersection
    exclusion: exclusion "\" intersection
    intersection: factor intersection | factor
    hull: factor "." hull | factor
    factor:
      interval | phase_linked_time_spec | event_linked_time_spec | "(" general_time_spec ")"
```

The Duration data value class provides for the specification of time intervals, and also for a simple string description of the periodicity.
9 Encapsulated Package

9.1 Overview

The data_types.encapsulated package contains classes representing data values whose internal structure is defined outside the EHR model, such as multimedia and parsable data. It is illustrated in FIGURE 11.

![FIGURE 11 rm.data_types.encapsulated Package](image)

9.1.1 Requirements

The requirement met by the encapsulated package is the need to be able to include content in the EHR whose interior structure is not modelled in the EHR reference model, but instead documented by sufficient meta-data attributes for specific tools to process the data. Types of content in this category are as follows.

- Any image, including images which are themselves a compressed version of one image from a high-resolution image set stored elsewhere. Such images may be in any of the well-known compressed or uncompressed formats, and may have their own thumbnail image attached, to facilitate web-viewing.
- Any bio-signal data series, such as a set of values representing a diagnostic part of an ECG trace. This might be represented as DICOM content.
- Any content which is textual (or nearly so) which is essentially a parsable language file of some kind. This includes all XML instance, HTML, and any other EHR content which happens to be represented in syntax form - such as the unit strings used in quantities. The name of the formalism should be stored as meta-data.
- Any binary content which is processed by a work processor or other dedicated tool.
- Digital signatures.
Sufficient meta-data must be included with all of these types to enable a way for the content to be processed, typically by indicating either its type (e.g. “jpeg”, “word document”) or the name of a tool which can be used to process it. Important meta-data include:

- size of the content;
- natural language, if any.

Any encapsulated data item may be a summary, “thumbnail” or otherwise reduced form of an original content item found outside the EHR, in some other system or file-system.

Checksums must be expressible for those items for which a checksum is available, or for which the system generates checksums to improve the quality of its internal data transmissions.

9.1.2 Design
The design approach used here is based on the following analysis.

1. Any encapsulated data item may be in some particular language, even if it is an image or other graphic form such as a biosignal with axis markings in a particular language;
2. The general structure of encapsulated content data items includes a block of bytes or characters representing the content, and various meta-data as appropriate, including:
   - size
   - character encoding
   - compression type/algorithm
   - name of formalism for parsable content
3. For encapsulated items that have a counterpart in another system, the standard means of portable address is the W3C URI;
4. For items that may have an associated integrity checksum, the checksum is itself a series of bytes, and the type of checksum must also be specified, e.g. “md5”.

These observations lead naturally to an abstract DV_ENCAPSULATED class, with two subtypes, DV_PARSABLE, for all content which is syntactic in nature, and DV_MULTIMEDIA for everything else. Note that it is possible to imagine parsable content items which are large, stored in compressed form, and are themselves a summary of another item elsewhere on the web; such items can for practical purposes be represented as instances of DV_MULTIMEDIA, rather than DV_PARSABLE. The vast majority of parsable encapsulated data are expected to be short and stored in native textual form, e.g. fragments of XML or HTML.

The formal model of the classes DV_ENCAPSULATED and DV_MULTIMEDIA are closely based on the ED type from the HL7v3 data types specification.

9.2 Class Descriptions

9.2.1 DV_ENCAPSULATED Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_ENCAPSULATED (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract class defining the common meta-data of all types of encapsulated data.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 2.6</td>
</tr>
<tr>
<td>CLASS</td>
<td><strong>DV_ENCAPSULATED</strong> (abstract)</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>CEN</td>
<td>TBD</td>
</tr>
<tr>
<td>OMG HDTF</td>
<td>COAS::MultiMedia</td>
</tr>
<tr>
<td>HL7</td>
<td>Encapsulated_data (ED)</td>
</tr>
<tr>
<td>Inherit</td>
<td>DATA_VALUE</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>charset:</td>
<td>CODE_PHRASE</td>
</tr>
<tr>
<td>language:</td>
<td>CODE_PHRASE</td>
</tr>
<tr>
<td>1..1</td>
<td>Name of language used if data is formatted text, from ISO 639:1988 (E/F) &quot;Code for the representation of names of languages&quot;. For a definitive rendition see <a href="http://www.unicode.org/unicode/onlinedat/languages.html">http://www.unicode.org/unicode/onlinedat/languages.html</a>. Otherwise Void.</td>
</tr>
<tr>
<td>size:</td>
<td>Integer</td>
</tr>
<tr>
<td>1..1</td>
<td>size in bytes of data. Note that for expanded data, size may not always equal <code>data.count</code></td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>as_string:</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>Result = alternate_text [(uri)]</td>
</tr>
</tbody>
</table>

**Invariant**

- Size_positive: size >= 0
- Language_valid: language /= Void and then code_set(“languages”).has(language)
- Charset_valid: charset /= Void and then code_set(“character sets”).has(charset)

### 9.2.2 DV_MULTIMEDIA Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th><strong>DV_MULTIMEDIA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A specialisation of DV_ENCAPSULATED for audiovisual and biosignal types. Includes further metadata relating to multimedia types which are not applicable to other subtypes of DV_ENCAPSULATED.</td>
</tr>
<tr>
<td>Use</td>
<td></td>
</tr>
<tr>
<td>ISO 18308</td>
<td>STR 3.1</td>
</tr>
</tbody>
</table>
### Synapses

The Bulky Data class provides for the representation and storage of all binary data classified by its MIME type.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternate_text</td>
<td>String</td>
<td>Text to display in lieu of multimedia display/replay</td>
</tr>
<tr>
<td>media_type</td>
<td>CODE_PHRASE</td>
<td>Data media type coded from the IANA MIME types code set. See: <a href="http://www.iana.org/assignments/media-types/">http://www.iana.org/assignments/media-types/</a></td>
</tr>
<tr>
<td>compression_algorithm</td>
<td>CODE_PHRASE</td>
<td>compression type, a coded value from the openEHR “Integrity check” code set. Void means no compression.</td>
</tr>
<tr>
<td>integrity_check</td>
<td>Array &lt;Character&gt;</td>
<td>binary cryptographic integrity checksum</td>
</tr>
<tr>
<td>integrity_check_algorithm</td>
<td>CODE_PHRASE</td>
<td>type of integrity check, a coded value from the openEHR “Integrity check” code set.</td>
</tr>
<tr>
<td>thumbnail</td>
<td>DV_MULTIMEDIA</td>
<td>the thumbnail for this item, if one exists; mainly for graphics formats.</td>
</tr>
<tr>
<td>uri</td>
<td>DV_URI</td>
<td>URI reference to electronic information stored outside the record as a file, database entry etc, if supplied as a reference.</td>
</tr>
<tr>
<td>data</td>
<td>Array &lt;Character&gt;</td>
<td>the actual data found at <code>uri</code>, if supplied inline</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_external</td>
<td>Boolean</td>
<td>Computed from the value of the <code>uri</code> attribute: True if the data is stored externally to the record, as indicated by <code>uri</code>. A copy may also be stored internally, in which case <code>is_expanded</code> is also true.</td>
</tr>
<tr>
<td>is_inline</td>
<td>Boolean</td>
<td>Computed from the value of the <code>data</code> attribute: True if the data is stored in expanded form, ie within the EHR itself.</td>
</tr>
</tbody>
</table>
### 9.2.3 DV_PARSABLE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_PARSABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Encapsulated data expressed as a parsable String. The internal model of the data item is not described in the openEHR model in common with other encapsulated types, but in this case, the form of the data is assumed to be plaintext, rather than compressed or other types of large binary data.</td>
</tr>
<tr>
<td>Use</td>
<td>Used for representing values which are formal textual representations, e.g. guidelines.</td>
</tr>
<tr>
<td>ISO 18308</td>
<td>(none)</td>
</tr>
<tr>
<td>Inherit</td>
<td>DV_ENCAPSULATED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>value: String</td>
<td>the string, which may validly be empty in some syntaxes</td>
</tr>
<tr>
<td>1..1</td>
<td>formalism: String</td>
<td>name of the formalism, e.g. “GLIF 1.0”, “proforma” etc.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariant</th>
<th>value_valid: value /= Void</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>formalism_validity: formalism /= Void and then not formalism.is_empty</td>
</tr>
</tbody>
</table>
10 Uri Package

10.1 Overview

The data_types.uri package includes two types used for referring to information resources. The DV_URI type allows data values which are references to objects on the world wide web to be created. Its specialisation, DV_EHR_URI, enables any element in an openEHR record to be identified in the same way as other objects on the web. The DV_EHR_URI type is convenient, because it is a string, like any other URI, and is therefore easily transportable and processable. Because it has its own scheme space, “ehr”, instances can be globally unique, as long as EHR identification is globally unique. DV_EHR URIs are used to express all runtime paths in the EHR. The uri Package is illustrated in FIGURE 12.

10.1.1 Requirements

This package meets the requirement for a DATA_VALUE subtype which represents a W3C Uniform Resource Identifier (URI). A common example of where this might be used is to represent a reference to a clinical guideline or other justifying document associated with an intervention or treatment plan recorded in the EHR.

URIs are a superset of Uniform Resource Locators (URLs) (although the two are often confused, even within the W3C), and can be used to specify the location of any information item, regardless of its type, location or storage method, as long as a URI “scheme” exists for that type of information.

There is an additional requirement for a kind of URI that can point at an EHR data item, either inside the same EHR containing the link, or in another EHR. This is the basis of implementing the LINK type.

10.1.2 Design

A simple design approach is used whereby a URI is represented as a String, and appropriate functions are defined to extract the various parts according to the syntax of URIs defined by Tim Berners-Lee at http://www.ietf.org/rfc/rfc2396.txt. An EHR specific subtype is defined, whose scheme is
“ehr”, and which contains further attributes enabling the instances of the type to record what kind of object they are referring to.

### 10.2 Definitions

The following symbolic definitions are used in the classes below.

- **Ehr_scheme**: String is “ehr”

### 10.3 Class Descriptions

#### 10.3.1 DV_URI Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>DV_URI</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>A reference to an object which conforms to the Universal Resource Identifier (URI) standard, as defined by W3C RFC 2936. See &quot;Universal Resource Identifiers in WWW&quot; by Tim Berners-Lee at <a href="http://www.ietf.org/rfc/rfc2396.txt">http://www.ietf.org/rfc/rfc2396.txt</a>. This is a World-Wide Web RFC for global identification of resources. See <a href="http://www.w3.org/Addressing">http://www.w3.org/Addressing</a> for a starting point on URIs. See <a href="http://www.ietf.org/rfc/rfc2806.txt">http://www.ietf.org/rfc/rfc2806.txt</a> for new URI types like telephone, fax and modem numbers.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Enables external resources to be referenced from within the content of the EHR. A number of functions return the logical subparts of the URI string.</td>
</tr>
<tr>
<td><strong>MisUse</strong></td>
<td>TBD</td>
</tr>
<tr>
<td><strong>CEN</strong></td>
<td>TBD</td>
</tr>
<tr>
<td><strong>OMG HDTF</strong></td>
<td>COAS::TechnologyInstanceLocator</td>
</tr>
<tr>
<td><strong>HL7</strong></td>
<td>TBD</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>DATA_VALUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>value: String</td>
<td>Value of URI as a String.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>


### scheme: String

A distributed information "space" in which information objects exist. The scheme simultaneously specifies an information space and a mechanism for accessing objects in that space. For example if scheme = "ftp", it identifies the information space in which all ftpable objects exist, and also the application - ftp - which can be used to access them. Values may include: "ftp", "telnet", "mailto", "gopher" and many others. Refer to WWW URI RFC for a full list.

New information spaces can be accommodated within the URI specification.

### path: String

A string whose format is a function of the scheme. Identifies the location in <scheme>-space of an information entity. Typical values include hierarchical directory paths for any machine. For example, with scheme = "ftp", path might be /pub/images/image_01. The strings "." and ".." are reserved for use in the path. Paths may include internet/intranet location identifiers of the form: sub_domain...domain, e.g. "info.cern.ch"

### fragment_id: String

A part of, a fragment or a sub-function within an object. Allows references to sub-parts of objects, such as a certain line and character position in a text object. The syntax and semantics are defined by the application responsible for the object.

### query: String

Query string to send to application implied by scheme and path. Enables queries to applications, including databases to be included in the URI. Any query meaningful to the server, including SQL.

#### Invariant

\[ value \neq Void \text{ and then not } value\.is\_empty \]

### 10.3.2 DV_EHR_URI Class

#### Purpose

A **DV_EHR_URI** is a **DV_URI** which has the scheme name “ehr”, and which can only reference elements in EHRs. The syntax is described below.

#### Use

Used to reference elements in an EHR, which may be the current one, or another.

#### Inherit

**DV_EHR_URI**
10.3.2.1 **DV_EHR_URI Syntax**

The syntax of a **DV_EHR_URI** is an *openEHR* path, inside the “ehr” URI scheme-space, and is of the form:

```
“ehr://” ehr_path
```

The syntax of ehr_path is described in the section on Paths in The *openEHR* Architecture Overview document. **DV_EHR_URIs** are used as a mechanism for referencing in the EHR, ensuring readability by humans, as well as validity when extracts are transmitted elsewhere: even if the target of a path is not present, the path can be used to locate the missing item on demand.
11 Implementation Strategies

11.1 Overview

This section notes a few of the general challenges for mapping the openEHR data types to implementation technologies such as programming languages and XML. For specific guidelines, Implementation Technology Specification (ITS) document for each target formalism should be consulted.

11.2 Quantities and Ordered_numeric

In the quantity package, the type DV_QUANTIFIED is shown having an abstract property of type Ordered_numeric. This is intended to indicate that the type DV_QUANTIFIED is distinguished by the magnitude property (compared to say DV_ORDERED, which describes ordered things without having magnitudes). The type Ordered_numeric be mapped to various types in implementation technologies as follows:

- Java: java.lang.Number
- C#: System.IComparable
- Eiffel: NUMERIC

All of these type systems currently suffer from not having a single type whose meaning is both “ordered” (having the function ‘<’) and “numeric” (having the functions ‘+’, ‘-’, ‘*’, '/') but in practice it does not matter much. For type systems with no convenient supertype of the numeric concrete types Real, Integer, Double, the magnitude property can safely be left out of DV_QUANTIFIED; the only drawback is that code cannot call DV_QUANTIFIED.magnitude polymorphically, e.g. in a statistical application processing DV_QUANTITY and DV_COUNT objects.

11.3 Unicode

Unicode is supported in various ways in different languages. In Java, since JDK 1.1, unicode support is implicit in the base classes. From the documentation:


In the C# language, conversion can be done between Unicode and other codepages using the System.Text.UnicodeEncoding (for UTF-16) and System.Text.UTF8Encoding (for UTF-8) classes.

In XML unicode is handled by specifying the encoding of the document in the XML declaration, e.g. <?xml version="1.0" encoding="UTF-16" ?>.

In the Eiffel language, unicode is available in the Gobo public domain library (see http://www.gobosoft.com), in the UC_STRING class, which inherits from the String class.

The support in other languages varies, and may require a special type like the UC_STRING used in Eiffel.
11.4 Dates and Times

In some formalisms, dates and times are represented using a single calendar-like class. This is not considered to be good practice from the point of specification, since it is more difficult to state proper invariants for such a class used to represent a particular logical type such as a DATE or TIME, however, its utility in implementation is recognised.

Where implementors want to use such a class (call it CALENDAR here for the sake of discussion) the recommended approach is to wrap the class CALENDAR with classes representing the types described in this specification, i.e. DATE etc. This enables the addition of any necessary functionality in the wrapper for example, for serialising and deserialising in and out of XML.
12 Comparison with HL7v3 Types

12.1 Scope
Some HL7v3 types are not modelled in openEHR. HL7v3 V3DT types which are assumed by openEHR to exist in the underlying type system of any implementation technology include:

- Integer (INT)
- Real (REAL)
- Set (SET)
- List (LIST)
- Bag (BAG)

HL7v3 types which are not modelled here because they are almost always too volatile for concrete modelling, and can be created with archetyped generic information structures are as follows (even in HL7 they are really data structures rather than data types):

- Postal address (AD)
- Entity name (EN)
- Person name (PN)
- Organisation name (ON)
- Trivial name (TN)

These types are all modelled by archetyped spatial data structures in openEHR (equivalent to sub-types of Structure in the CDA specification).

HL7v3 types which may need to be modelled in the future include:

- Uncertain value probabilistic (UVP)
- Non-parametric probability distribution (NPPD)
- Parametric probability distribution (PPD)

Types which are provided by openEHR but not supported directly by HL7 include:

- state variable (DV_STATE);
- ordinal values (DV_ORDINAL);
- explicit partial date and time types (DV_PARTIAL_DATE, DV_PARTIAL_TIME);
- explicit time duration (DV_DURATION).

Types in the latter two categories appear to be implementable with the TS (timestamp) type.

12.2 Design Differences
There are some significant differences in design approach between the openEHR data types and the HL7v3 data types, described in the following sections.

12.2.1 Naming
All types in the HL7 specification have two names, one short and one long. For example the type representing physical quantities is known both as “PhysicalQuantity” and “PQ”. While short names may be reasonable for often-used types, someshort names are not obvious, e.g. “EN”, “ON”, “TN”, “NPPD” etc. Short names certainly have benefits for drawing tools such as Rational Rose or other
Comparison with HL7v3 Types

Rev 2.0

The openEHR model uses one name only for each class.

12.2.2 Identification

The HL7 V3DT includes the types II, UID, OID and UUID. The II type is claimed to be for identifying all kinds of entities, which we here classify as real-world entities (“RWEs”) (such as people, vehicle registrations, invoices) and informational entities (“IEs”) - which in general are snapshots of data representing an RWE in a computer system. One problem with RWE identification schemes is that some are known (e.g. social security number) to produce fallible identifiers or situations where multiple RWEs have the same identifier, or no identifier at all. Conversely, with well-controlled and internationally agreed ways of issuing/generating information system identifiers (e.g. GUID, ISO OID) it is thought that such identifiers can be made reliable, and indeed correspond 1:1 with their intended IEs. However, a problem with IEs is that there are often duplicates and also multiple versions in time, each intended to represent the same RWE (such as a particular person, observation or composition).

As far as can be ascertained currently, there is no standard analysis taking into account the existence of IEs and RWEs, and recognising the fact that multiple versions and/or duplicates may refer to the same RWE.

The approach taken in openEHR with respect to identifiers is currently as follows.

- RWE identifiers such as social security numbers, licence numbers, etc are modelled with the type `DV_IDENTIFIER`, which has the attributes:
  - issuer: String
  - id: String
  - type: String

  The attributes listed above are nearly the same as for the HL7 II type, indicating that the two types may be compatible.

- Identification of IEs is done using the type `OBJECT_ID`, which is not a data type, and is documented in the Support Information Model. The `OBJECT_ID` type takes into account the fact that there may be multiple IEs referring to the same underlying RWE by adding a version identifier (assumed to be a timestamp).

12.2.3 Archetyping

The openEHR data types are defined on the assumption of archetype-based systems. While they will work perfectly well in systems which know nothing about archetyping, some types are not defined because archetypable structures built from more basic entities are assumed instead, rather than concretely modelled data types. These include “types” for address and person name which are found in HL7v3 and CEN 13606.

12.2.4 Treatment of Inbuilt Types

The HL7v3 data types do not make any assumptions about the existence of types typically built-in to most object and relational formalisms, such as the basic types `String`, `Integer`, `Boolean`, `Real`, `Double`, and the generic types `Set<T>`, `Bag<T>` and `Array<T>`. Hence, the types `ST`, `INT`, `REAL`, `BL`, `SET<T>`, `BAG<T>` and so on are redefined by HL7. The supposed advantage of this approach is that the semantics of all types in the HL7 system, right down to atomic data items are self-contained.
in the definition, and do not rely on external semantics. Possible problems with this approach include
the following.

- The HL7 definitions diverge from the OMG IDL and ISO 11404 definitions of the basic data
types, which could cause unexpected problems in software development and data processing
which is done in typical development technologies (object-oriented and relational).
- The HL7 types INT and REAL are defined as subtypes of the QTY type, a relationship that
does not exist in any object-oriented formalism for these types (in particular, there is no sub-
stitutability of a type called Integer or Real for a type called Qty built in to any object lan-
guage). The definitions of INT and REAL are also different from those found in most object
formalisms. This might cause some difficulty in implementation.
- The binary data type BIN is represented as a List<BL> (where each item can be True,
False, null), whereas it would normally be expected to be something like Array<Charac-
ter> (i.e. an array of bytes) in most software environments. There does not appear to be any
utility in defining it as List<BL>, since binary data is almost without exception represented
and processed as contiguous arrays of machine bytes.
- The string type ST inherits from the encapsulated data type ED, which in turn inherits from
the binary data type BIN. The result of this is that an instance of ST contains numerous data
attributes relating to multi-media data, and the content is presumably represented as a
List<BL>. This is a major departure from the standard understanding of a string in compu-
ter sciences, which is usually simply an array of characters.
- The HL7 boolean type BL is a three-valued logic type due to the null marker approach (see
below), not the usual two-valued type found in the Boolean concept in programming lan-
guages. The same is true of INT and REAL: due to the null marker design, “null” is a possi-
ble return value of an integer or real as well as true integer and real values.

In general, where differences exist between same-named types in HL7 and an underlying formalism
such as a programming language, there is likely to be some confusion in implementation. Further,
there is likely to be confusion in how to process instances of basic types which contain numerous (and
sometimes recursive) fields which are not used in the standard specifications of basic types.

The openEHR approach with respect to inbuilt types is to assume only those types found in the main-
stream object-oriented programming languages, and in particular, definitive formalisms like OMG
IDL and XML. While this means there there is in theory less control over these types than in the HL7
approach, the number of types involved is quite small, and the problem of bindings to the basic types
of object formalisms is well understood. Additionally, since it is recognised that some data types
defined by openEHR could clash with types found in some languages and libraries, all data type class
names are prefaced with “DV_” to avoid naming confusion, and to allow implementations of
openEHR types to co-exist with existing types in implementation formalisms.

### 12.2.5 Use of Null Markers

All HL7 data types inherit from the ANY class (equivalent to the DATA_VALUE class in openEHR)
which contains the attributes:

```java
BL nonNull;
CS nullFlavor;
BL isNull;
```

The purpose of these attributes is to indicate whether a datum is Null, and for what reason. Since
some data type classes also appear as the attributes of other data types, the Null markers also indicate
whether any part of a datum is null. Thus, in the class \texttt{Interval<T>} shown below, all attributes have the possibility of containing a Null marker.

\begin{verbatim}
type Interval<T> alias IVL<T> extends Set<T> {
    T low;
    BL lowClosed;
    T high;
    BL highClosed;
    T.diff width;
    T.center;
    IVL<T> hull(IVL<T> x);
    literal ST;
    promotion IVL<T> (T x);
    demotion T;
}
\end{verbatim}

For example, this allows an interval with missing ends and width to exist as a structured type. The consequence of the approach is that the entire model is essentially a model of “partial” data types; \textit{any attribute and any function call may return a Null value, as well as the true values of its type} (in fact, in the specification, Null values are defined to be valid values of all data types). This design decision was taken in HL7 so that any datum, no matter how unknown, would be structurally representable in the same way as completely known data, enabling it to be processed in the same way as all other instances of the same type.

However, an important object-oriented design principle has been ignored in this approach. In the proper design of classes, properties and \textit{class invariants} are stated. Invariants are statements which describe the correctness conditions of instances of the class; the general rule is that the post-condition of a creation routine (constructor) of a class must be that the invariants are satisfied. For example, an invariant of the HL7 \texttt{IVL<T>} class could be:

\begin{verbatim}
(exists(low) and exists(high)) or else
(exists(low) and exists(width)) or else
(exists(width) and exists(high))
\end{verbatim}

When an instance of this class is created, this condition should be satisfied, and remain satisfied for the life of the instance. To do otherwise is to create instances of data which other software can make no assumptions about, and is forced to check every single field, and then determine what to do in an \textit{ad hoc} way. (See [6] p366, [4] p43, [5] p29 for detailed explanations of the invariant concept).

Possible consequences of the built-in Null marker design approach include:

- since even HL7’s basic types \texttt{ST, INT, REAL, LIST<>}, \texttt{SET<>} include null markers, processing of null values will be pervasive at the lowest level;
- software will be more complex, both implementations of the data types, and of software which handle them. This is because the software always has to deal with the possibility of calls to routines and attributes returning Null values. Most clinical information systems to date have taken the approach that a datum is either represented as an instance of a formal type if fully known, or else as narrative text if only partial;
- data may not be always be safely processable, since some software may not properly handle the null values associated with attributes of partially known data items. Essentially, all software which processes the data has to be “null-value aware”, and make no assumptions at all about whether a particular data instance is valid or not.

The HL7 data type model is in contrast with simpler approaches such as used in CEN, GEHR, and openEHR, where data types are formal models of types such as \texttt{Coded_term}, \texttt{Quantity} and so on.
Rather than build the possibility of null markers into every attribute and class in the data types, a single null marker is defined in relevant containing classes. This decision is based on the principle that data types should be defined independently of their context of use. Hence, where data types are used as data values, such as in the value attribute of the class ELEMENT from the openEHR EHR reference model, the parallel features is_null and null_flavour are also defined. However, where data types appear as attributes elsewhere in the model and there is no possibility of them being null, no null markers are used. FIGURE 13 shows visually the difference between the two approaches.

The consequences of the standard software-engineering approach include:

- data types can be more easily formally specified, since the semantics of invariants, attributes and operations do not need to include the possibility of null values;
- software implementations are simpler;
- data are always guaranteed to be safely processable for decision support and general querying, since no instance of a formal type will be created in the first place if the datum is very unreliable;
- null markers only appear in models where they are relevant, rather than everywhere data types are used;
- however, the openEHR data types do not automatically deal with missing or unknown internal attribute values (such as missing high and low values for an interval, partial dates etc).

In order to deal with the last possibility, various approaches are used in openEHR:

- for most data which is not fully known, no data type instance is created, and a null marker is created. Depending on the design of the relevant archetypes, there will usually be the possibility of recording the datum in narrative form;
- ENTRY-s in the openEHR EHR reference model include a certainty: Boolean attribute, for recording a level of doubt;
Comparison with HL7v3 Types

The openEHR Data Types Information Model
Rev 2.0

- for particular data types which are often partial, special types are defined. The main types affected are `DV_DATE` and `DV_TIME`, hence the types `DV_PARTIAL_DATE` and `DV_PARTIAL_TIME` exist to define explicitly the semantics of dates with a missing day, times with missing seconds and so on;

- Intervals of date/time types include types generated when the parameter type is one of the partial classes, thus, types `DV_INTERVAL<DV_PARTIAL_DATE>` are possible. This covers the need for intervals in which some date is missing from the end date/times, while not allowing intervals with completely missing items to be created;

- for expressing uncertainty more precisely, probability distribution data types (based on the types defined in HL7) can be used.

A consequence of the HL7 model is that data instances represented in XML or another structured text format will be structurally the same regardless of whether there are Null values or not. A structured form for partially known data (which would normally break the invariants of its class) may well be useful for representing the data as part of a text field, making it easier to use for whatever processing is possible later on.

12.2.6 Terminology Approach

The approach in openEHR is to assume the existence of a Terminology Server which is the sole authoritative interface with terminologies of any kind, and is the only entity which can assume responsibility for querying, post-coordination or other manipulations of terms. No allowance is made for coordination of “modifiers”, “qualifiers” or any other terms outside the service. As a consequence, there are no coordination facilities in the type `DV_CODED_TEXT`, a departure from earlier versions of the specification - any term provided from the terminology service must already be “coordinated”, either by the terminology service, or by one of the terminologies it accesses. This places the responsibility of combining terms firmly in the knowledge part of the system, and prevents unsanctioned, unvalidated combinations being created elsewhere.

12.2.7 Date/Time Approach

The HL7 specification uses a single TS type to represent all logical dates, times, date/times, and partial versions thereof. The openEHR specification defines distinct types for each, since these are the types which occur in the real world, and it is easier to specify correctness constraints with this approach. It is recognised that a single type may be used by some implementors (depending on what is available in the language being used), however, the recommended practice is to wrap any such types with the logical types described in this specification. This approach reduces the possibility for any errors in transmitted data (since no strange combinations of year, ..., second can occur not explicitly described in the type definitions).

12.2.8 Time Specification Types

The HL7 approach for time specification appears to cover all reasonable requirements, but has some minor problems, including:

- the types `PIVL` and `EIVL` are declared as being generic types (i.e. `PIVL<T:TS>`, `EIVL<T:TS>`), when there appears to be no reason for this;

- the `PIVL.phase` attribute is used to represent an interval during which a activity occurs, example given is "2 minutes every 8 hours". However, the "2 mins" is almost always part of a therapeutic prescription of some kind, not part of the timing specification as such. Therapeutic prescriptions have the form "do X every Y time", where the X describes what to do, and how long to do it for (e.g. 40 mins massage, administer a drug slowly over 10 mins). In
fact, what we are really interested in with a timing specification is the specification of the *starting points in time* of some activity, not a time-based graph of on/off points, which is effectively what the PIVL type is now.

### 12.2.9 Type Conversions

The HL7v3 data types specification allows various type conversions, as follows:

Three kinds of type conversions are defined: promotion, demotion, and character string literals. Type conversions can be implicit or explicit. Implicit type conversion occurs when a certain type is expected (e.g. as an argument to a statement) but a different type is actually provided.

One notable kind of conversion possible in HL7 is of a value of any type $T$ into an instance of $\text{Set}<T>$, $\text{List}<T>$, $\text{Bag}<T>$ or $\text{IVL}<T>$ containing the value.

The openEHR model does not provide for any type conversions other than those automatically available between inbuilt basic numeric types such as Integer, Float and Double, and between types related by inheritance, as supported by all object-oriented languages.
A References

A.1 General
1 Berners-Lee T. "Universal Resource Identifiers in WWW". Available at http://www.ietf.org/rfc/rfc2396.txt. This is a World-Wide Web RFC for global identification of resources. In current use on the web, e.g. by Mosaic, Netscape and similar tools. See http://www.w3.org/Addressing for a starting point on URIs.

A.2 European Projects

A.3 CEN
11 ENV 13606-1 - Electronic healthcare record communication - Part 1: Extended architecture. CEN/ TC 251 Health Informatics Technical Committee.
12 ENV 13606-2 - Electronic healthcare record communication - Part 2: Domain term list. CEN/ TC 251 Health Informatics Technical Committee.
13 ENV 13606-3 - Electronic healthcare record communication - Part 3: Distribution rules. CEN/ TC 251 Health Informatics Technical Committee.

A.4 GEHR Australia
A.5  HL7

REFERENCE MODEL

The openEHR Support Information Model

Editors: {T Beale, S Heard}¹, {D Kalra, D Lloyd}²

Revision: 1.5

Pages: 43

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.9</td>
<td>CR-000063. ATTESTATION should have a status attribute.</td>
<td>D Kalra</td>
<td>13 Feb 2004</td>
</tr>
<tr>
<td>0.9.8</td>
<td>CR-000068. Correct errors in INTERVAL class.</td>
<td>T Beale</td>
<td>20 Dec 2003</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>0.9.7</td>
<td>CR-000032. Basic numeric type assumptions need to be stated.</td>
<td>DSTC, D Lloyd, T Beale</td>
<td>09 Oct 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000041. Visually differentiate primitive types in openEHR documents.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000043. Move External package to Common RM and rename to Identification (incorporates CR-000036 - Add HIER_OBJECT_ID class, make OBJECT_ID class abstract.)</td>
<td>T Beale</td>
<td>18 Sep 2003</td>
</tr>
<tr>
<td>0.9.6</td>
<td>CR-000013. Rename key classes. Based on CEN ENV13606.</td>
<td>T Beale</td>
<td>16 Aug 2003</td>
</tr>
<tr>
<td></td>
<td>CR-000038. Remove archetype_originator from multi-axial archetype id.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR-000039. Change archetype_id section separator from ':' to '-'.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.9.5</td>
<td>CR-000036. Add HIER_OBJECT_ID class, make OBJECT_ID class abstract.</td>
<td>T Beale</td>
<td></td>
</tr>
<tr>
<td>0.9.4</td>
<td>CR-000022. Code TERM_MAPPING.purpose.</td>
<td>G Grieve</td>
<td>20 Jun 2003</td>
</tr>
<tr>
<td>0.9.3</td>
<td>CR-000007. Added forgotten terminologies for Subject_relationships and Provider_functions.</td>
<td>T Beale</td>
<td>11 Apr 2003</td>
</tr>
<tr>
<td>0.9.2</td>
<td>Detailed review by Ocean, DSTC, Grahame Grieve. Updated valid characters in OBJECT_ID.namespace.</td>
<td>G Grieve</td>
<td>25 Mar 2003</td>
</tr>
<tr>
<td>0.9.1</td>
<td>Added specification for BOOLEAN type. Corrected minor error in ISO 639 standard strings - now conformant to TERMINOLOGY_ID. OBJECT_ID.version_id now optional. Improved document structure.</td>
<td>T Beale</td>
<td>18 Mar 2003</td>
</tr>
<tr>
<td>0.9</td>
<td>Initial Writing. Taken from Data types and Common Reference Models. Formally validated using ISE Eiffel 5.2.</td>
<td>T Beale</td>
<td>25 Feb 2003</td>
</tr>
</tbody>
</table>

**Acknowledgements**

The work reported in this paper has been funded by a number of organisations, including The University College, London and Ocean Informatics, Australia.
Table of Contents

1 Introduction .............................................................................. 7
  1.1 Purpose............................................................................................... 7
  1.2 Related Documents ........................................................................... 7
  1.3 Status.................................................................................................... 7
  1.4 Peer review .......................................................................................... 7
  1.5 Conformance.......................................................................................... 7

2 Support Package ...................................................................... 9
  2.1 Overview ............................................................................................... 9
  2.2 Class Definitions ................................................................................. 9
    2.2.1 EXTERNAL_ENVIRONMENT_ACCESS Class .................................. 9

3 Assumed Types ........................................................................11
  3.1 Overview ............................................................................................... 11
  3.2 Inbuilt Primitive Types .......................................................................... 12
    3.2.1 Any Type .......................................................................................... 13
    3.2.2 Boolean Type .................................................................................. 13
    3.2.3 Ordered_numeric Type .................................................................... 14
  3.3 Assumed Library Types ......................................................................... 15
    3.3.1 String Type ..................................................................................... 16
    3.3.1.1 UNICODE ........................................................................................ 16
    3.3.2 Aggregate Type .............................................................................. 17
    3.3.3 Hash Type ...................................................................................... 17
  3.4 Date/Time Types ................................................................................. 17
    3.4.1 Interval Type .................................................................................. 18

4 Identification Package ........................................................... 19
  4.1 Overview ............................................................................................... 19
  4.1.1 Requirements .................................................................................. 19
  4.1.2 Identifying Real World Entities (RWE) ........................................... 21
  4.1.3 Identifying Informational Entities (IEs) .......................................... 21
  4.1.4 Identifying Versions of Informational Entities .............................. 22
  4.1.5 Referring to Informational Entities ................................................. 22
  4.2 Class Descriptions............................................................................... 22
    4.2.1 OBJECT_REF Class ........................................................................ 22
    4.2.2 ACCESS_GROUP_REF Class ............................................................. 23
    4.2.3 PARTY_REF Class .......................................................................... 23
    4.2.4 LOCATABLE_REF Class ................................................................. 24
    4.2.5 OBJECT_ID Class ........................................................................... 24
    4.2.6 HIER_OBJECT_ID Class ................................................................. 25
      4.2.6.1 Identifier Syntax ............................................................................. 25
    4.2.7 OBJECT_VERSION_ID Class ........................................................... 25
      4.2.7.1 Identifier Syntax ............................................................................. 26
    4.2.8 VERSION_TREE_ID Class ............................................................... 26
    4.2.9 ARCHETYPE_ID Class ..................................................................... 27
      4.2.9.1 Archetype ID Syntax ..................................................................... 28
    4.2.10 TERMINOLOGY_ID Class ............................................................. 29
      4.2.10.1 Identifier Syntax ........................................................................... 29
    4.2.11 UID Class ...................................................................................... 30

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
The openEHR Support Information Model

4.2.12 ISO_OID Class .......................................................... 30
4.2.13 UUID Class .......................................................... 30
4.2.14 INTERNET_ID Class .............................................. 31

5 Terminology Package .................................................. 33
  5.1 Overview ................................................................. 33
  5.2 Service Interface ..................................................... 33
    5.2.1 Class Definitions .................................................. 34
      5.2.1.1 TERMINOLOGY_SERVICE Class ......................... 34
      5.2.1.2 TERMINOLOGY_ACCESS Class ............................ 34
      5.2.1.3 CODE_SET_ACCESS Class ................................. 35

6 Measurement Package ................................................. 37
  6.1 Overview ............................................................... 37
  6.2 Service Interface ..................................................... 37
    6.2.1 Class Definitions .................................................. 37
      6.2.1.1 MEASUREMENT_SERVICE_ACCESS Class .............. 37

7 Definition Package ..................................................... 39
  7.1 Overview ............................................................... 39
    7.1.1 Class Definitions .................................................. 39
      7.1.1.1 OPENEHR_DEFINITIONS Class ............................. 39

A References ............................................................... 41
  A.1 General ................................................................. 41
1 Introduction

1.1 Purpose
This document describes the openEHR Support Reference Model, whose semantics are used by all openEHR Reference Models. The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development organisations developing EHR systems;
- Academic groups studying the EHR;
- The open source healthcare community.

1.2 Related Documents
Prerequisite documents for reading this document include:

- The openEHR Modelling Guide

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

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

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

Blue text indicates sections under active development.

1.4 Peer review
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.

1.5 Conformance
Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Support Package

2.1 Overview

The Support Reference Model comprises types which are used throughout other openEHR models, but are defined elsewhere, either by standards organisations or which are accepted de facto standards. The package structure is illustrated in FIGURE 1.

![Diagram showing the relationship between support, terminology, measurement, identification, assumed_types, and EXTERNAL_ENVIRONMENT_ACCESS classes.]

FIGURE 1 rm.support and assumed_types Packages

The four Support packages define the semantics respectively for constants, terms, scientific measurement and identifiers, which are assumed by the rest of the openEHR specifications. The class EXTERNAL_ENVIRONMENT_ACCESS is a mixin class providing access to external services.

2.2 Class Definitions

2.2.1 EXTERNAL_ENVIRONMENT_ACCESS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXTERNAL_ENVIRONMENT_ACCESS (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A mixin class providing access to services in the external environment.</td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td>eea_terminology_svc: TERMINOLOGY_SERVICE</td>
<td>Return an interface to the terminology service</td>
</tr>
<tr>
<td>eea_measurement_svc: MEASUREMENT_SERVICE</td>
<td>Return an interface to the measurement service</td>
</tr>
<tr>
<td>CLASS</td>
<td>EXTERNAL_ENVIRONMENT_ACCESS (abstract)</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
<tr>
<td>Terminology_service_exists: eea_terminology_svc /= Void</td>
<td></td>
</tr>
<tr>
<td>Measurement_service_exists: eea_measurement_svc /= Void</td>
<td></td>
</tr>
</tbody>
</table>
3 Assumed Types

3.1 Overview

This section describes types assumed by all openEHR models. The set of types chosen here is based on a lowest common denominator set from threes sources, as follows.

- Well-known interoperability formalisms, including OMG IDL, W3C XML-schema.
- Well-known object-oriented programming languages, including C++, Java, C#, and Eiffel.

The intention in openEHR is to make the minimum possible assumptions about types found in implementation formalisms, while making sufficient assumptions to both enable openEHR models to be conveniently specified, and to allow the typical basic types of these formalisms to be used in their normal way, rather than being re-invented by openEHR. The ISO 11404 (2003) standard contains basic semantics of “general purpose data types” (GPDs) for information technology, and is used here as a normative basis for describing assumptions about types. The operations and properties described here are compatible with those used in ISO 11404, but not always the same, as 11404 has not chosen to use object-oriented functions. For example, the notional function \texttt{has}(x:T) (test for presence of a value in a set) defined on the type \texttt{Set<T>} below is not defined on the ISO 11404 Set type; instead, the function \texttt{IsIn}(x: T; s: Set<T>) is defined. However, in object-oriented formalisms, the function \texttt{IsIn} defined on a Set type would usually mean “subset of”, i.e. true if this set is contained inside another set. In the interests of clarity for developers, an object-oriented style of functions and properties has been used here.

Two groups of assumed types are identified: primitive types, which are those built in to a formalism’s type system, and library types, which are assumed to be available in a (class) library defined in the formalism. Thus, the type \texttt{Boolean} is always assumed to exist in a formalism, while the type \texttt{Array<T>} is assumed to be available in a library. For practical purposes, these two categories do not matter that much - whether \texttt{String} is really a library class (the usual case) or an inbuilt type doesn’t make much difference to the programmer. They are shown separately here mainly as an explanatory convenience.

The assumptions that openEHR makes about existing types are documented below in terms of interface definitions. Each of these definitions contains only the assumptions required for the given type to be used in the openEHR Reference Model - it is not by any means a complete interface definition. The name and semantics of any function used here for an assumed type might not be identical to those found in some implementation technologies, but should be very close. Any mapping required should be stated in the relevant ITS. The definitions are compatible with the ISO 11404 standard, 2003 revision. Operation semantics are described formally using pre- and post-conditions. The keyword “Current” stands for “the current instance” (known as “this” or “self” in various languages). The keyword “like” anchors the type of the reference to the type of the object whose reference follows like. Not all types have definition tables - only those which add features to their inheritance parent have a table.
### 3.2 Inbuilt Primitive Types

The following types constitute the minimum built in set of types assumed by openEHR of an implementation formalism.

<table>
<thead>
<tr>
<th>Type name in openEHR</th>
<th>Description</th>
<th>ISO 11404 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>represents a type whose value is a member of an 8-bit character-set (ISO: “repertoire”).</td>
<td>Character</td>
</tr>
<tr>
<td>Boolean</td>
<td>represents logical True/False values; usually physically represented as an integer, but need not be</td>
<td>Boolean</td>
</tr>
<tr>
<td>Integer</td>
<td>represents 32-bit integers</td>
<td>Integer</td>
</tr>
<tr>
<td>Real</td>
<td>represents 32-bit real numbers in any interoperable representation, including single-width IEEE floating point</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>type which represents 64-bit real numbers, in any interoperable representation including double-precision IEEE floating point.</td>
<td>Real</td>
</tr>
</tbody>
</table>

As shown in the table, openEHR assumes that Character is an 8-bit type. This is because the only use of Character in openEHR is in encapsulated data (openEHR Data Types), where the intention is to represent opaque data. Note that “octet” would probably be a more correct name to use here, but it generally is not used in programming languages.

FIGURE 2 illustrates the inbuilt types. Simple inheritance relationships are shown which facilitate the type descriptions below. A class “Any” is therefore used to stand for the usual top-level class in all object-oriented type systems, typically called something like “Any” or “Object”. Inheritance from or substitutability for an Any class is not assumed at all in openEHR (hence the dotted lines in the UML). It is used to enable basic operations like ‘=’ to be described once for the type Any, rather than in every subtype. The type Ordered_numeric is on the other hand assumed for purposes of specification in the openEHR data_types.quantity package, and is intended to be mapped to an equivalent type in a real type system (e.g. in Java, java.lang.Number). Here it is assumed that the operations defined on Ordered_numeric are available on the types Integer, Real and Double in implementation type systems, where relevant. Data-oriented implementation type systems such as XML-schema are not expected to have such operations.

![FIGURE 2 Primitive Types Assumed by openEHR](image-url)
### 3.2.1 Any Type

<table>
<thead>
<tr>
<th>INTERFACE</th>
<th>Any (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Abstract supertype. Usually maps to a type like “Any” or “Object” in an object system. Defined here to provide the value and reference equality semantics.</td>
</tr>
<tr>
<td>Abstract</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>is_equal (other: Any): Boolean</td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>infix ‘=’ (other: Any): Boolean</td>
</tr>
<tr>
<td></td>
<td>type: String</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>

### 3.2.2 Boolean Type

<table>
<thead>
<tr>
<th>INTERFACE</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Boolean type used for two-valued mathematical logic.</td>
</tr>
<tr>
<td>Abstract</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>infix &quot;and&quot; (other: Boolean): Boolean</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>other_exists: other /= void</td>
</tr>
<tr>
<td></td>
<td>ensure</td>
</tr>
<tr>
<td></td>
<td>de_morgan: Result = not (not Current or not other)</td>
</tr>
<tr>
<td></td>
<td>commutative: Result = (other and Current)</td>
</tr>
<tr>
<td></td>
<td>infix &quot;and then&quot; (other: Boolean): Boolean</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>other_exists: other /= void</td>
</tr>
<tr>
<td></td>
<td>ensure</td>
</tr>
<tr>
<td></td>
<td>de_morgan: Result = not (not Current or else not other)</td>
</tr>
</tbody>
</table>
### 3.2.3 Ordered_numeric Type

<table>
<thead>
<tr>
<th>INTERFACE</th>
<th>Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>infix &quot;or&quot; (other: Boolean): Boolean</td>
<td></td>
</tr>
<tr>
<td>require</td>
<td></td>
</tr>
<tr>
<td>other_exists: other /= void</td>
<td></td>
</tr>
<tr>
<td>ensure</td>
<td></td>
</tr>
<tr>
<td>de_morgan: Result = not (not Current and not other)</td>
<td></td>
</tr>
<tr>
<td>commutative: Result = (other or Current)</td>
<td></td>
</tr>
<tr>
<td>consistent_with_semi_strict: Result implies (Current or else other)</td>
<td></td>
</tr>
<tr>
<td>infix &quot;or else&quot; (other: Boolean): Boolean</td>
<td></td>
</tr>
<tr>
<td>require</td>
<td></td>
</tr>
<tr>
<td>other_exists: other /= void</td>
<td></td>
</tr>
<tr>
<td>ensure</td>
<td></td>
</tr>
<tr>
<td>de_morgan: Result = not (not Current and then not other)</td>
<td></td>
</tr>
<tr>
<td>infix &quot;xor&quot; (other: Boolean): Boolean</td>
<td></td>
</tr>
<tr>
<td>require</td>
<td></td>
</tr>
<tr>
<td>other_exists: other /= void</td>
<td></td>
</tr>
<tr>
<td>ensure</td>
<td></td>
</tr>
<tr>
<td>definition: Result = ((Current or other) and not (Current and other))</td>
<td></td>
</tr>
<tr>
<td>infix &quot;implies&quot; (other: Boolean): Boolean</td>
<td></td>
</tr>
<tr>
<td>require</td>
<td></td>
</tr>
<tr>
<td>other_exists: other /= void</td>
<td></td>
</tr>
<tr>
<td>ensure</td>
<td></td>
</tr>
<tr>
<td>definition: Result = (not Current or else other)</td>
<td></td>
</tr>
</tbody>
</table>

#### Invariants

- involutive_negation: is_equal (not (not Current))
- non_contradiction: not (Current and (not Current))
- completeness: Current or else (not Current)

### INTERFACE

#### Ordered_numeric (abstract)

**Purpose**

Abstract notional parent class of ordered, numeric types, which are types which have various arithmetic and comparison operators defined. All ordered, quantified types (i.e. types with a notion of precise “magnitude”) have these operations. Maps to various types in implementation technologies.

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
### Assumed Library Types

The types described in this section are also assumed to be fairly standard by *openEHR*, but usually to come from type libraries rather than be built into the type system of implementation formalisms.

<table>
<thead>
<tr>
<th>Type name in openEHR</th>
<th>Description</th>
<th>ISO 11404: 2003 Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>represents unicode-enabled strings</td>
<td>Character-String/Sequence</td>
</tr>
<tr>
<td>Array&lt;T&gt;</td>
<td>physical container of items indexed by number</td>
<td>Array</td>
</tr>
<tr>
<td>List&lt;T&gt;</td>
<td>container of items, implied order, non-unique membership</td>
<td>Sequence</td>
</tr>
<tr>
<td>Set&lt;T&gt;</td>
<td>container of items, no order, unique membership</td>
<td>Set</td>
</tr>
<tr>
<td>Bag&lt;T&gt;</td>
<td>container of items, no order, non-unique membership</td>
<td>Bag</td>
</tr>
</tbody>
</table>

### 3.3 Assumed Library Types

The types described in this section are also assumed to be fairly standard by *openEHR*, but usually to come from type libraries rather than be built into the type system of implementation formalisms.
Assumed Types

Assumed Types The openEHR Support Information Model
Rev 1.5

FIGURE 3 illustrates the assumed library types. As with the assumed primitive types, inheritance and abstract classes are used for convenience of the definitions below, but are not formally assumed in openEHR.

3.3.1 String Type

<table>
<thead>
<tr>
<th>INTERFACE</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Strings of characters, as used to represent textual data in any natural or formal language.</td>
</tr>
<tr>
<td>Functions</td>
<td></td>
</tr>
<tr>
<td>Signature</td>
<td>Meaning</td>
</tr>
<tr>
<td>infix ‘+’ (other: String): String</td>
<td>Concatenation operator - causes ‘other’ to be appended to this string</td>
</tr>
<tr>
<td>is_empty: Boolean</td>
<td>True if string is empty, i.e. equal to “”</td>
</tr>
<tr>
<td>is_integer: Boolean</td>
<td>True if string can be parsed as an integer</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>

3.3.1.1 UNICODE

It is assumed in the openEHR specifications that Unicode is supported by the type String. Unicode is needed for all Asian, Arabic and other script languages, for both data values (particularly plain text and coded text) and for many predefined string attributes of the classes in the openEHR Reference Model. It encompasses all existing character sets.
### 3.3.2 Aggregate Type

**INTERFACE**

**Aggregate <T> (abstract)**

<table>
<thead>
<tr>
<th>Description</th>
<th>Abstract parent of the aggregate types List&lt;T&gt;, Set&lt;T&gt;, Bag&lt;T&gt;, Array&lt;T&gt; and Hash&lt;T,K&gt;.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>has (v: T):</td>
<td>Boolean</td>
<td>Test for membership of a value</td>
</tr>
<tr>
<td>count:</td>
<td>Integer</td>
<td>Number of items in container</td>
</tr>
</tbody>
</table>

| Invariants  |                                                             |

### 3.3.3 Hash Type

**INTERFACE**

**Hash <T, U: Comparable>**

<table>
<thead>
<tr>
<th>Description</th>
<th>Type representing a keyed table of values. T is the value type, and U the type of the keys.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>has_key (a_key: U):</td>
<td>Boolean</td>
<td>Test for membership of a key</td>
</tr>
<tr>
<td>item (a_key: U):</td>
<td>T</td>
<td>Return item for key ‘a_key’. Equivalent to ISO 11404 fetch operation.</td>
</tr>
</tbody>
</table>

| Invariants  |                                                             |

### 3.4 Date/Time Types

Although the ISO 11404 (2003) standard defines a date-and-time type generator (section 8.1.6), and a `timeinterval` type (section 10.1.6), the reality is that dates and times are provided in significantly differing ways in implementation formalisms, and as a result, openEHR assumes nothing at all about them. Accordingly, types for date, time, date/time and duration are defined in the openEHR Data Types Information Model, ensuring standardised meanings of these types within openEHR. ISO 8601 is used as the normative basis for both string literal representation and properties chosen within these models.
3.4.1 Interval Type

<table>
<thead>
<tr>
<th>INTERFACE</th>
<th>Interval &lt;T:Ordered&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Interval of ordered items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower: T</td>
<td></td>
<td>lower bound</td>
</tr>
<tr>
<td>upper: T</td>
<td></td>
<td>upper bound</td>
</tr>
<tr>
<td>lower_unbounded: Boolean</td>
<td></td>
<td>lower boundary open (i.e. = -infinity)</td>
</tr>
<tr>
<td>upper_unbounded: Boolean</td>
<td></td>
<td>upper boundary open (i.e. = +infinity)</td>
</tr>
<tr>
<td>lower_included: Boolean</td>
<td></td>
<td>lower boundary value included in range if not lower_unbounded</td>
</tr>
<tr>
<td>upper_included: Boolean</td>
<td></td>
<td>upper boundary value included in range if not upper_unbounded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>has(e:T): Boolean</td>
<td></td>
<td>True if (lower_unbounded or ((lower_included and v &gt;= lower) or v &gt; lower)) and (upper_unbounded or ((upper_included and v &lt;= upper or v &lt; upper)))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Limits_consistent:</td>
<td>(not upper_unbounded and not lower_unbounded) implies lower &lt;= upper</td>
</tr>
<tr>
<td>Limits_comparable:</td>
<td>(not upper_unbounded and not lower_unbounded) implies lower.strictly_comparable_to(upper)</td>
</tr>
</tbody>
</table>
4 Identification Package

4.1 Overview

The identification package describes a model of references and identifiers for information entities only and is illustrated in FIGURE 4. Real-world entity identifiers are defined in the openEHR Data Types information model.

4.1.1 Requirements

Identification of entities both in the real world and in information systems is a non-trivial problem. The scenarios for identification across systems in a health information environment include the following:

- real world identifiers such as social security numbers, veterans affairs ids etc can be recorded as required by health care facilities, enterprise policies, or legislation;
- identifiers for informational entities which represent real world entities or processes should be unique;
- it should be possible to determine if two identifiers refer to information entities that are linked to the same real world entity, even if instances of the information entities are maintained in different systems;

FIGURE 4 rm.support.identification Package
Identification Package

The openEHR Support Information Model

Rev 1.5

• versions or changes to real-world entity-linked informational entities (which may create new information instances) should be accounted for in two ways:
  - it should be possible to tell if two identifiers refer to distinct versions of the same informational entity in the same version tree;
  - it should not be possible to confuse same-named versions of informational entities maintained in multiple systems which purport to represent the same real world entity. E.g. there is no guarantee that two systems’ “latest” version of the Person “Dr Jones” is the same.

Medico-legal use of information relies on previous states of information being identifiable in some way.

• it should be possible for an entity in one system or service (such as the EHR) to refer to an entity in another system or service in such a way that:
  - the target of the reference is easily finable within the shared environment, and
  - the reference does is valid regardless of the physical architecture of servers and applications.

The following subsections describe some of the features and challenges of identification.

Identification of Real World Entities (RWEs)

Real world entities such as people, car engines, invoices, and appointments all have identifiers. Although many of these are designed to be unique within a jurisdiction, they are often not, due to data entry errors, bad design (ids which are too small or incorporate some non-unique characteristic of the identified entities), bad process (e.g. non-synchronised id issuing points); identity theft (e.g. via theft of documents of proof or hacking). In general, while some real world identifiers (RWIs) are “nearly unique”, none can be guaranteed so. It should also be the case that if two RWE identifiers are equal, they refer to the same RWE.

Identification of Informational Entities (IEs)

As soon as information systems are used to record facts about RWEs, the situation becomes more complex because of the intangible nature of information. In particular:

• the same RWE can be represented simultaneously on more than one system (“spatial multiplicity”);
• the same RWE may be represented by more than one “version” of the same IE in a system (“temporal multiplicity”).

At first sight, it appears that there can also be purely informational entities, i.e. IEs which do not refer to any RWE, such as books, online-only documents and software. However, as soon as one considers an example it becomes clear that there is always a notional “definitive” or “authoritative” (i.e. trusted) version of every such entity. These entities can better be understood as “virtual RWEs”. Thus it can still be said that multiple IEs may refer to any given RWE.

The underlying reason for the multiplicity of IEs is that “reality” - time and space - in computer systems is not continuous but discrete, and each “entity” is in fact just a snapshot of certain attribute values of a RWE.

If identifiers are assigned to IEs without regard to versions or duplicates, then no assertion can be made about the identified RWE when two IE ids are compared.
Identification of Versions of Informational Entities

The notion of “versioning” applies only to informational entities, i.e. distinct instances of content each representing a snapshot of some notional information. Where such instances are stored and managed in versioned containers, within a versioning system of some kind, explicit identification of the versions is required. The requirements are discussed in detail in the Common IM, change_control package. They can be summarised as follows:

- it must be possible to distinguish two versions of the same notional entity, i.e. know from the identifier if they are the same or different versions of the same thing;
- it must be possible to tell the relationship between the items in a versioned lineage, from the version identifiers.

Referencing of Informational Entities

Within a distributed information environment, there is a need for entities not connected by direct references in the same memory space to be able to refer to each other. There are two competing requirements:

- that the separation of objects in a distributed computing environment not compromise the semantics of the model. At the limit, this mandates the use of proxy types which have the same abstract interface as the proxied type; i.e. the “static” approach of Corba.
- that different types of information can be managed relatively independently; for example EHR and demographic information can be managed by different groups in an organisation or community, each with at least some freedom to change implementation and model details.

4.1.2 Identifying Real World Entities (RWE)

In openEHR, Real world entities are identified with a multipart identifier expressed in the data type DV_IDENTIFIER. This type should be used to express lab result identifiers, veterans affairs numbers and so on, i.e. any identifier issued by an organisation and corresponding to a continuant (an entity that continues to exist even if its attributes change over time).

4.1.3 Identifying Informational Entities (IEs)

The class OBJECT_ID is an abstract model of identifiers of IEs. It is assumed a priori that there can in general be more than one IE referring to the same underlying real world entity (RWE), such as a person or invoice; this is due to the possible existence of multiple copies, and also multiple versions. An OBJECT_ID therefore implicitly refers to a version of something; two versions of a Person object must have two distinct OBJECT_IDs. The rule for versioning is that if any attribute value of the IE changes, a new OBJECT_ID must be generated. Some OBJECT_ID subtypes explicitly model a version identifier. In practice, it can usually be omitted for ids of terminologies, where the terminology obeys the rule that a given code never changes its meaning through all versions of the terminology (i.e. ICD10 code F40.0 will mean “Agoraphobia” for all time (in English)).

The subtype HIER_OBJECT_ID defines a hierarchical identifier model, along the lines of ISO Oids; it includes the attributes root and extension, to make up a complete, unique identifier. The root attribute is of type UID, meaning it has the properties of a timeless unique object identifier. Subtypes of UID include the ISO_OID and UUID types. The latter models a DCE UUID (also known as a GUID).

The other subtypes, ARCHETYPE_ID and TERMINOLOGY_ID define different kinds of identifier, the former being a multi-axial identifier for archetypes, and the latter being a globally unique single string identifier for terminologies.
4.1.4 Identifying Versions of Informational Entities

The scheme used in openEHR for identifying versions uses a three-part identifier, consisting of:

- the identifier of the version container, in the form of an OBJECT_ID;
- the location in the version tree, as a 1- or 3-part numeric identifier, where the latter type expresses branching;
- the identifier of the system in which this version was created.

Under this scheme, multiple versions in the same container all have the same value for the first identifier, while their location in the version tree is given by the combination of the version tree identifier and the identifier of the creating system.

The format of the creating_system_id attribute is not currently fixed, hence its type is HIER_OBJECT_ID, allowing for various possibilities. The requirements on this identifier are that it be unique per system, and that it be easy to obtain or generate. It is also helpful if it is a meaningful identifier. The two most practical candidates appear to be GUIDs (which are not meaningful, but are easy to generate) and reverse internet domain identifiers, as recommended in [3] (these are easy to determine if the system has an internet address, and are meaningful and directly processible, however unconnected systems pose a problem). ISO Oids might also be used. All of these identifier types are accommodated via the use of HIER_OBJECT_ID.

A full explanation of the version identification scheme and its capabilities is given in the change_control section of the Common IM.

4.1.5 Referring to Informational Entities

All OBJECT_IDs are used as identifier attributes within the thing they identify, in the same way as a database primary key. To refer to an identified object, an instance of the class OBJECT_REF is required, in the same way as a database foreign key. OBJECT_REF is provided as a means of distributed referencing, and includes the object namespace (typically 1:1 with some service, such as “terminology”) and type. The general principle of object references is to be able to refer to an object available in a particular namespace or service. Usually they are used to refer to objects in other services, such as a demographic entity from within an EHR, but they may be used to refer to local objects as well. The type may be the concrete type of the referred-to object (e.g. “GP”) or any proper ancestor (e.g. “PARTY”). The notion of object reference provided here is a compromise between the static binding notion of Corba (where each model is dependent on all the interface details of the classes in other models) and a purely dynamic referencing scheme, where the holder of a reference cannot even tell what type of object the reference points to.

4.2 Class Descriptions

4.2.1 OBJECT_REF Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OBJECT_REF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Class describing a reference to another object, which may exist locally or be maintained outside the current namespace, e.g. in another service. Services are usually external, e.g. available in a LAN (including on the same host) or the internet via Corba, SOAP, or some other distributed protocol. However, in small systems they may be part of the same executable as the data containing the Id.</td>
</tr>
</tbody>
</table>
### 4.2.2 ACCESS_GROUP_REF Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>OBJECT_ID</td>
<td>Globally unique id of an object, regardless of where it is stored.</td>
</tr>
<tr>
<td>namespace</td>
<td>String</td>
<td>Namespace to which this identifier belongs in the local system context (and possibly in any other openEHR compliant environment) e.g. “terminology”, “demographic”. These names are not yet standardised. Legal values for the namespace are “local”</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Name of the class of object to which this identifier type refers, e.g. “PARTY”, “PERSON”, “GUIDELINE” etc. These class names are from the relevant reference model. The type name “ANY” can be used to indicate that any type is accepted (e.g. if the type is unknown).</td>
</tr>
</tbody>
</table>

**Invariant**

- **Id_exists**: id /= Void
- **Namespace_exists**: namespace /= Void and then not namespace.empty
- **Type_exists**: type /= Void and then not type.empty

### 4.2.3 PARTY_REF Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Identifier for parties in a demographic or identity service. There are typically a number of subtypes of the PARTY class, including PERSON, ORGANISATION, etc.</td>
</tr>
</tbody>
</table>

**Inherit**

- OBJECT_REF

**Functions**

- **Type_validity**: type.is_equal(“ACCESS_GROUP”)

---

Editors: (T Beale, S Heard), (D Kalra, D Lloyd)  
Page 23 of 43  
Date of Issue: 06 Feb 2006  
© 2003-2006 The openEHR Foundation  
email: info@openEHR.org  
web: http://www.openEHR.org
### 4.2.4 LOCATABLE_REF Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>LOCATABLE_REF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Reference to a LOCATABLE instance inside the top-level content structure inside a VERSION&lt;T&gt;; the path attribute is applied to the object that VERSION.data points to.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>OBJECT_REF</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td><strong>id</strong>: OBJECT_VERSION_ID</td>
</tr>
<tr>
<td>1..1 (redefined)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>path</strong>: String</td>
</tr>
<tr>
<td></td>
<td><strong>as_uri</strong>: String</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td><em>Path_valid</em>: path /= Void implies not path.is_empty</td>
</tr>
</tbody>
</table>

### 4.2.5 OBJECT_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OBJECT_ID (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Ancestor class of identifiers of informational objects. Ids may be completely meaningless, in which case their only job is to refer to something, or may carry some information to do with the identified object.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Object_ids are used inside an object to identify that object. To identify another object in another service, use an OBJECT_REF, or else use a UID for local objects identified by UID.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td><strong>value</strong>: String</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td><em>Value_exists</em>: value /= Void and then not value.empty</td>
</tr>
</tbody>
</table>
4.2.6  HIER_OBJECT_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>HIER_OBJECT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Hierarchical identifiers consisting of a root part and an optional extension.</td>
</tr>
<tr>
<td>HL7</td>
<td>The HL7v3 II Data type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>UID</td>
<td>The identifier of the conceptual namespace in which the object exists, within the identification scheme.</td>
</tr>
<tr>
<td>has_extension</td>
<td>Boolean</td>
<td>True if there is an extension part.</td>
</tr>
<tr>
<td>extension</td>
<td>String</td>
<td>A local identifier of the object within the context of the root identifier.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariant</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Root_valid</td>
<td>root /= Void</td>
</tr>
<tr>
<td>Extension_valid</td>
<td>extension /= Void</td>
</tr>
</tbody>
</table>

4.2.6.1  Identifier Syntax

The syntax of the value attribute by default follows the following pattern:

<root> "::" <extension>

4.2.7  OBJECT_VERSION_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OBJECT_VERSION_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Globally unique identifier for one version of a versioned object.</td>
</tr>
<tr>
<td>Inherit</td>
<td>OBJECT_ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1</td>
<td>object_id: UID</td>
<td>Unique identifier for logical object of which this identifier identifies one version; normally the object_id will be the unique identifier of the version container containing the version referred to by this OBJECT_VERSION_ID instance.</td>
</tr>
<tr>
<td>1..1</td>
<td>version_tree_id: VERSION_TREE_ID</td>
<td>Tree identifier of this version with respect to other versions in the same version tree, as either 1 or 3 part dot-separated numbers, e.g. “1”, “2.1.4”.</td>
</tr>
<tr>
<td>1..1</td>
<td>creating_system_id: HIER_OBJECT_ID</td>
<td>Identifier of the system that created the Version corresponding to this Object version id.</td>
</tr>
</tbody>
</table>
### 4.2.7.1 Identifier Syntax

The string form of an `OBJECT_VERSION_ID` consists of three segments separated by double colons ("::"), i.e.:

```
<object_id>::<creating_system_id>::<version_tree_id>
```

An example is as follows:

```
F7C5C7B7-75DB-4b39-9A1E-C0BA9BFDBDEC::87284370-2D4B-4e3d-A3F3-F303D2F4F34B::2
```

### 4.2.8 VERSION_TREE_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VERSION_TREE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Version tree identifier for one version.</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: String</td>
<td>String form of this identifier. Format is: <code>&lt;trunk_version&gt;[.&lt;branch_number&gt;.&lt;branch_version&gt;]</code>.</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>trunk_version: String</td>
<td>Trunk version number.</td>
</tr>
<tr>
<td>branch_number: String</td>
<td>Number of branch from the trunk point.</td>
</tr>
<tr>
<td>branch_version: String</td>
<td>Version of the branch.</td>
</tr>
<tr>
<td>is_branch: Boolean</td>
<td>True if this version identifier represents a branch, i.e. has <code>branch_number</code> and <code>branch_version</code> parts.</td>
</tr>
</tbody>
</table>
### 4.2.9 ARCHETYPE_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Identifier for archetypes.</td>
</tr>
<tr>
<td>Inherit</td>
<td>OBJECT_ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>qualified_rm_entity: String</td>
<td></td>
<td>Globally qualified reference model entity, e.g. “openehr-ehr_rm-entry”.</td>
</tr>
<tr>
<td>domain_concept: String</td>
<td></td>
<td>Name of the concept represented by this archetype, including specialisation, e.g. “biochemistry result-cholesterol”.</td>
</tr>
<tr>
<td>rm_originator: String</td>
<td></td>
<td>Organisation originating the reference model on which this archetype is based, e.g. “openehr”, “cen”, “hl7”.</td>
</tr>
<tr>
<td>rm_name: String</td>
<td></td>
<td>Name of the reference model, e.g. “rim”, “ehr_rm”, “en13606”.</td>
</tr>
<tr>
<td>rm_entity: String</td>
<td></td>
<td>Name of the ontological level within the reference model to which this archetype is targeted, e.g. for openEHR, “folder”, “composition”, “section”, “entry”.</td>
</tr>
<tr>
<td>specialisation: String</td>
<td></td>
<td>Name of specialisation of concept, if this archetype is a specialisation of another archetype, e.g. “cholesterol”.</td>
</tr>
<tr>
<td>version_id: String</td>
<td></td>
<td>Version of this archetype.</td>
</tr>
</tbody>
</table>
### 4.2.9.1 Archetype ID Syntax

Archetype identifiers are “multi-axial”, meaning that each identifier instance denotes a single archetype within a multi-dimensional space. In this case, the space is essentially a versioned 3-dimensional space, with the dimensions being:

- reference model entity, i.e. target of archetype
- domain concept
- version

As with any multi-axial identifier, the underlying principle of an archetype id is that all parts of the id must be able to be considered immutable. This means that no variable characteristic of an archetype (e.g. accrediting authority, which might change due to later accreditation by another authority, or may be multiple) can be included in its identifier. The syntax of an **ARCHETYPE_ID** is as follows:

```
archetype_id: qualified_rm_entity '.' domain_concept '.' version_id

qualified_rm_entity: rm_originator '-' rm_name '-' rm_entity
rm_originator: NAME
rm_name: NAME
rm_entity: NAME

domain_concept: concept_name { '-' specialisation }*
concept_name: NAME
specialisation: NAME

version_id: 'v' NUMBER

NUMBER: [0-9]*
NAME: [a-z][a-z0-9/]%$#&]*
```

The field meanings are as follows:

- **rm_originator**: id of organisation originating the reference model on which this archetype is based;
- **rm_name**: id of the reference model on which the archetype is based;
- **rm_entity**: ontological level in the reference model;
- **domain_concept**: the domain concept name, including any specialisations;
- **version_id**: numeric version identifier;

Examples of archetype identifiers include:

**Invariant**

```
Qualified_rm_entity_valid: qualified_rm_entity /= Void and then not qualified_rm_entity.is_empty
Domain_concept_valid: domain_concept /= Void and then not domain_concept.is_empty
Rm_originator_valid: rm_originator /= Void and then not rm_originator.is_empty
Rm_name_valid: rm_name /= Void and then not rm_name.is_empty
Rm_entity_valid: rm_entity /= Void and then not rm_entity.is_empty
Specialisation_valid: specialisation /= Void implies not specialisation.is_empty
Version_id_valid: version_id /= Void and then not version_id.is_empty
```
Archetypes can also be identified by other means, such as ISO oids.

### 4.2.10 TERMINOLOGY_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TERMINOLOGY_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Identifier for terminologies such accessed via a terminology query service. In this class, the value attribute identifies the Terminology in the terminology service, e.g. “SNOMED-CT”. A terminology is assumed to be in a particular language, which must be explicitly specified. The value if the id attribute is the precise terminology id identifier, including actual release (i.e. actual “version”), local modifications etc; e.g. “ICPC2”</td>
</tr>
<tr>
<td>Inherit</td>
<td>OBJECT_ID</td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>name: String</td>
</tr>
<tr>
<td></td>
<td>version_id: String</td>
</tr>
<tr>
<td>Invariants</td>
<td>Name_valid: name /= Void and then not name.is_empty</td>
</tr>
<tr>
<td></td>
<td>Version_id_valid: version_id /= Void</td>
</tr>
</tbody>
</table>

#### 4.2.10.1 Identifier Syntax

The syntax of the value attribute is as follows:

```
name [ "(" version ")" ]
```

Examples of terminology identifiers include:

- “snomed-ct”
- “ICD9(1999)”

Versions should only be needed for those terminologies which break the rule that the thing being identified with a code loses or changes its meaning over versions of the terminology. This should not be the case for well known modern terminologies and ontologies, particularly those designed since the publication of Cimino’s ‘desiderata’ [1] of which the principle of “concept permanance” is applicable here - “A concept's meaning cannot change and it cannot be deleted from the vocabulary”. However, there maybe older terminologies, or specialised terminologies which may not have obeyed these rules, but which are still used; version ids should always be used for these.
### 4.2.11 UID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>UID (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent of classes representing unique identifiers which identify information entities in a durable way. UIDs only ever identify one IE in time or space and are never re-used.</td>
</tr>
<tr>
<td>HL7</td>
<td>The HL7v3 UID Data type.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: String</td>
<td>The value of the id.</td>
<td></td>
</tr>
</tbody>
</table>

| Invariant | Value_exists: value /= Void and then not value.empty |

### 4.2.12 ISO_OID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ISO_OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model of ISO’s Object Identifier (oid) as defined by the standard ISO/IEC 8824. Oids are formed from integers separated by dots. Each non-leaf node in an Oid starting from the left corresponds to an assigning authority, and identifies that authority’s namespace, inside which the remaining part of the identifier is locally unique.</td>
</tr>
<tr>
<td>HL7</td>
<td>The HL7v3 OID Data type.</td>
</tr>
<tr>
<td>Inherit</td>
<td>UID</td>
</tr>
</tbody>
</table>

### 4.2.13 UUID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>UUID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model of the DCE Universal Unique Identifier or UUID which takes the form of hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by the Open Group, CDE 1.1 Remote Procedure Call specification, Appendix A. Also known as a GUID.</td>
</tr>
<tr>
<td>HL7</td>
<td>The HL7v3 UUID Data type.</td>
</tr>
<tr>
<td>Inherit</td>
<td>UID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>
### 4.2.14 INTERNET_ID Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>INTERNET_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model of a reverse internet domain, as used to uniquely identify an internet domain. In the form of a dot-separated string in the reverse order of a domain name, specified by IETF RFC 1034 (<a href="http://www.ietf.org/rfc/rfc1034.txt">http://www.ietf.org/rfc/rfc1034.txt</a>).</td>
</tr>
<tr>
<td>Inherit</td>
<td>UID</td>
</tr>
<tr>
<td>Functions</td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>Invariant</td>
<td></td>
</tr>
</tbody>
</table>
5 Terminology Package

5.1 Overview
This section describes the terminology package, which contains classes for accessing the openEHR support terminology from within instances of classes defined in the reference model.

5.2 Service Interface
A simple terminology service interface is defined according to FIGURE 5, enabling openEHR terms to be referenced formally from within the Reference Model.

![FIGURE 5 rm.support.terminology Package]

Structural attributes in the Reference Model, such as FEEDER_AUDIT:change_type are defined by an invariant in the enclosing class, such as the following:

*Change_type_valid: terminology(“openehr”).codes_for_group_name(“audit change type”, “en”).has(change_type.defining_code)*

This is a formal way of saying that the attribute change_type must have a value such that its defining_code (its CODE_PHRASE) is in the set of CODE_PHRASEs in the openEHR Terminology which are in the group called (in English) “audit change type”.

A similar invariant is used for attributes of type CODE_PHRASE, which come from a code_set:

*Media_type_terminology: media_type /= Void and then code_set(“media types”).all_codes.has(media_type)
5.2.1 Class Definitions

5.2.1.1 TERMINOLOGY_SERVICE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TERMINOLOGY_SERVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Defines an object providing proxy access to a terminology service.</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>terminology (name: String):</td>
<td>TERMINOLOGY_ACCESS</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>has_terminology (name: String)</td>
</tr>
<tr>
<td></td>
<td>Result /= Void</td>
</tr>
<tr>
<td>code_set (name: String):</td>
<td>CODE_SET_ACCESS</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>has_code_set (name: String)</td>
</tr>
<tr>
<td></td>
<td>Result /= Void</td>
</tr>
<tr>
<td>has_terminology (name: String):</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>name.is_empty</td>
</tr>
<tr>
<td>has_code_set (name: String):</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td>require</td>
</tr>
<tr>
<td></td>
<td>name.is_empty</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>

5.2.1.2 TERMINOLOGY_ACCESS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TERMINOLOGY_ACCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Defines an object providing proxy access to a terminology.</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>id: String</td>
<td></td>
</tr>
</tbody>
</table>
## 5.2.1.3 CODE_SET_ACCESS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TERMINOLOGY_ACCESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_codes: Set&lt;CODE_PHRASE&gt;</td>
<td>Return all codes known in this terminology</td>
</tr>
<tr>
<td>codes_for_group_id (group_id: String): Set&lt;CODE_PHRASE&gt;</td>
<td>Return all codes under grouper ‘group_id’ from this terminology</td>
</tr>
<tr>
<td>codes_for_group_name (name, lang: String): Set&lt;CODE_PHRASE&gt;</td>
<td>Return all codes under grouper whose name in ‘lang’ is ‘name’ from this terminology</td>
</tr>
<tr>
<td>rubric_for_code (code, lang: String): String</td>
<td>Return all rubric of code ‘code’ in language ‘lang’</td>
</tr>
</tbody>
</table>

**Invariants**  
:id_exists: id /= Void and then not id.is_empty

### Purpose
 Defines an object providing proxy access to a code_set.

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: String</td>
<td>Identification of this Terminology</td>
</tr>
<tr>
<td>all_codes: Set&lt;CODE_PHRASE&gt;</td>
<td>Return all codes known in this terminology</td>
</tr>
<tr>
<td>has_lang (a_lang: CODE_PHRASE): Boolean</td>
<td>True if code set knows about ‘a_lang’</td>
</tr>
<tr>
<td>has_code (a_code: CODE_PHRASE): Boolean</td>
<td>True if code set knows about ‘a_code’</td>
</tr>
</tbody>
</table>

**Invariants**  
:id_exists: id /= Void and then not id.is_empty

---

© 2003-2006 The openEHR Foundation  
email: info@openEHR.org web: http://www.openEHR.org
6 Measurement Package

6.1 Overview
The Measurement package defines a minimum of semantics relating to quantitative measurement, units, and conversion, enabling the Quantity package of the openEHR Data Types Information Model to be correctly expressed. As for the Terminology package, a simple service interface is assumed, which provides useful functions to other parts of the reference model. The definitions underlying measurement and units come from a variety of sources, including:

- CEN ENV 12435, Medical Informatics - Expression of results of measurements in health sciences (see http://www.centc251.org);
- the Unified Code for Units of Measure (UCUM), developed by Gunther Schadow and Clement J. McDonald of The Regenstrief Institute (available in HL7v3 ballot materials; http://www.hl7.org).

These of course rest in turn upon a vast amount of literature and standards, mainly from ISO on the subject of scientific measurement.

6.2 Service Interface
A simple measurement data service interface is defined according to FIGURE 6, enabling quantitative semantics to be used formally from within the Reference Model. Note that this service as currently defined in no way seeks to properly model the semantics of units, conversions etc - it provides only the minimum functions required by the openEHR Reference Model.

![FIGURE 6 rm.support.measurement Package](image_url)

6.2.1 Class Definitions

6.2.1.1 MEASUREMENT_SERVICE_ACCESS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>MEASUREMENT_SERVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Defines an object providing proxy access to a measurement information service.</td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td><code>is_valid_units_string (units: String): Boolean</code></td>
</tr>
<tr>
<td></td>
<td><code>units_equivalent (units1, units2: String): Boolean</code></td>
</tr>
<tr>
<td>CLASS</td>
<td>MEASUREMENT_SERVICE</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td><strong>units_equivalent</strong> (units1, units2: String): Boolean</td>
</tr>
<tr>
<td></td>
<td><strong>require</strong></td>
</tr>
<tr>
<td></td>
<td>units1 /= Void and then</td>
</tr>
<tr>
<td></td>
<td>is_valid_units_string(units1)</td>
</tr>
<tr>
<td></td>
<td>units2 /= Void and then</td>
</tr>
<tr>
<td></td>
<td>is_valid_units_string(units2)</td>
</tr>
<tr>
<td></td>
<td>True if two units strings correspond to the same measured property.</td>
</tr>
</tbody>
</table>

**Invariants**
7 Definition Package

7.1 Overview

The definition package, illustrated in FIGURE 7, describes symbolic definitions used by the openEHR models.

7.1.1 Class Definitions

7.1.1.1 OPENEHR_DEFINITIONS Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OPENEHR_DEFINITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Defines an object providing proxy access to a measurement information service.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>CR: Character is ‘\015’</td>
<td>Carriage return character</td>
</tr>
<tr>
<td>LF: Character is ‘\012’</td>
<td>Linefeed character</td>
</tr>
<tr>
<td>Invariants</td>
<td></td>
</tr>
</tbody>
</table>
A References

A.1 General

END OF DOCUMENT
REFERENCE MODEL

The openEHR Integration Information Model

Editors: T Beale

Revision: 0.5

Pages: 15

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman: David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members: Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>Initial Writing</td>
<td>T Beale</td>
<td>15 Sep 2005</td>
</tr>
</tbody>
</table>

**Acknowledgements**

The work reported in this paper has been funded by The University College, London and Ocean Informatics, Australia.
# Table of Contents

1 Introduction .............................................................................. 7  
1.1 Purpose...................................................................................................7  
1.2 Related Documents ..............................................................................7  
1.3 Status......................................................................................................7  
1.4 Peer review ............................................................................................7  
1.5 Conformance..........................................................................................7  

2 Integration Package .................................................................................. 9  
2.1 Requirements .........................................................................................9  
2.2 Design Basis ..........................................................................................9  
2.2.1 Overview..........................................................................................9  
2.2.2 Semantics of GENERIC_ENTRY .................................................10  
2.2.3 Integration with CEN EN13606 ....................................................10  
2.3 Class Descriptions................................................................................11  
2.3.1 GENERIC_ENTRY Class .............................................................11  

A References ............................................................................... 13  
A.1 Standards..............................................................................................13
1 Introduction

1.1 Purpose
This document describes the architecture of the openEHR Integration Information Model, designed for use in legacy and other integration situations.

The intended audience includes:

- Standards bodies producing health informatics standards;
- Software development groups using openEHR;
- Academic groups using openEHR;
- 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 openEHR Reference Model documents

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

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

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

1.4 Peer review
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.

1.5 Conformance
Conformance of a data or software artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Integration Package

2.1 Requirements

Getting data in and out of the EHR is one of the most basic requirements openEHR aims to satisfy. In “greenfield” (new build) situations, and for data being created by GUI applications via the openEHR EHR APIs, there is no issue, since native openEHR structures and semantics are being used. In almost all other situations, existing data sources and sinks have to be accounted for. In general, external or ‘legacy’ data (here the term is used for convenience, and does not imply anything about the age or quality of the systems in question) have different syntactic and semantic formats than openEHR data, and seamless conversion requires addressing both levels.

Typical examples of legacy data sources and sinks include relational databases and HL7v2 messages. HL7v2 messages are probably one of the most common sources of pathology messages in many countries; EDIFACT messages are another. More recently, HL7v2 messages have been designed for referrals and even discharge summaries. Not all legacy systems are standardised; many if not most hospitals as well as GP and other desktop products have their own private models of data and terminology usage. Technically speaking, there is not much difference between standardised and non-standardised legacy models; only the reusability of the solution differs.

One important category of externally sourced data addressed by the Integration package described here is data expressed in the form of a CEN EN13606 Extract. Part 1 of EN13606 defines a information model which is nearly identical to that of openEHR at the COMPOSITION and SECTION levels. The CEN EN13606 Entry class is a generic structure with a minimum of contextual meta-data, and can easily be mapped to the openEHR Entry type described in this specification.

2.2 Design Basis

2.2.1 Overview

The design basis for connecting legacy systems to openEHR is founded upon a clear separation of the syntactic and semantic transformations required on data. The syntactic transformation converts source data from its original form (or whatever intermediate form it may have been converted to) to a format obeying a special class in the openEHR reference model, but whose logical structure and semantics are controlled by ‘legacy’ archetypes so as to mimic the design of the source data. This step brings the data into the openEHR computational context. The second step causes transformation on this intermediate openEHR data into data which are a) instances of the main openEHR reference model, and b) obey ‘designed’ clinical archetypes.

The additional elements of the openEHR architecture which make this transformation possible are:

- a class GENERIC_ENTRY, which is a sibling of SECTION and ENTRY, and contains completely generic, archetypable structures;
- ‘legacy’ archetypes, i.e. archetypes defined against the GENERIC_ENTRY class;
- semantic transformation rules from openEHR data based on GENERIC_ENTRY and legacy archetypes to data based on the subtypes of ENTRY, and designed archetypes.

FIGURE 1 illustrates the rm.integration package, which contains a single class GENERIC_ENTRY. Unlike other classes in the openEHR reference model, GENERIC_ENTRY contains no hard-wired attributes at all, only one generic attribute, data. No assumptions at all are made about the actual shape of such data.
2.2.2 Semantics of GENERIC_ENTRY

A number of useful consequences follow from this modelling approach. Firstly, instances of GENERIC_ENTRY will contain attributes inherited from the LOCATABLE class, including archetype_node_id, and are thus archetypable in the same way as all other classes in the openEHR reference model. The LOCATABLE attribute feeder_audit is also inherited, and may be used to mark every node of data with relevant meta-data from the source system record or message. Secondly, as a subtype of CONTENT_ITEM, GENERIC_ENTRY is a valid value for COMPOSITION.content. This is a completely desirable situation, since the same rules apply to GENERIC_ENTRY as to other content: instances can only be committed to the record as part of a COMPOSITION instance. GENERIC_ENTRY data are thus audit-trailed and versioned in the normal way. Thirdly, GENERIC_ENTRY instances can occur within a hierarchy of SECTIONS, which is useful for data sources which have headings or section equivalents (this is quite common in hospital information systems containing physician notes). Lastly, in common with all other openEHR data, design-time paths can be constructed for archetypes of GENERIC_ENTRY, while runtime paths can be extracted from data based on such archetypes. These path sets can be used for writing the data transformation rules.

It should be remembered that while GENERIC_ENTRY provides a standardised syntactic form for externally sourced data within openEHR, it provides no semantic coherence. This is particularly true for GENERIC_ENTRY instances sourced from numerous data sources: there is no guarantee that the GENERIC_ENTRY representations of “cholesterol result” from system A will be congruent with those sourced from system B. It is not even required that the data sources be vastly different for this problem to occur. Examples of messages can be found coming from different pathology laboratories, which obey the same minor version of HL7v2 (e.g. 2.3.1) and supposedly implement the same message type (e.g. “complete blood picture”) but which differ in actual structure and content.

The consequence of this situation is that GENERIC_ENTRY data cannot in general safely be used for clinical computation (e.g. decision support), and will not in general even support reliable clinical querying. In other words, a repository of GENERIC_ENTRYs (within appropriate COMPOSITION structures) does not constitute a reliable or interoperable health record - it can only be considered a standardised health information data store whose primary purpose is as the input to or output of semantic conversion processes, or for other auditing or non-clinical data management purposes.

2.2.3 Integration with CEN EN13606

The GENERIC_ENTRY class provides a convenient basis for making openEHR systems EN13606-compliant, which in turn gives openEHR a gateway capability in heterogeneous environments where
EN13606 is being used to communicate data. A CEN EN13606 EHR Extract can be converted to a series of **COMPOSITIONs** containing **GENERIC_ENTRY** objects which obey appropriate legacy archetypes; this data can then be semantically converted into orthodox openEHR objects for integration into a coherent EHR. Similarly, openEHR data can be converted into the **GENERIC_ENTRY**-based intermediate form for further conversion into EN13606 EHR Extracts.

### 2.3 Class Descriptions

#### 2.3.1 GENERIC_ENTRY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>GENERIC_ENTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>This class is used to create intermediate representations of data from sources not otherwise conforming to openEHR classes, such as HL7 messages, relational databases and so on.</td>
</tr>
<tr>
<td><strong>CEN</strong></td>
<td>Entry</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>CONTENT_ITEM</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong>: ITEM_TREE</td>
<td></td>
<td>The ‘data’ from the source message or record.</td>
</tr>
</tbody>
</table>
A.1 Standards

1. ENV 13606-1 - *Electronic healthcare record communication - Part 1: Extended architecture.*
   CEN/ TC 251 Health Informatics Technical Committee.

2. HL7 version 2 ref....
END OF DOCUMENT
Archetype Definitions and Principles

Editors: {T Beale, S Heard}¹

Revision: 0.6

Pages: 13

Keywords: EHR, archetypes, constraints

---

1. Ocean Informatics Australia

© 2003-2006 The openEHR Foundation

The openEHR foundation

is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

**Founding Chairman**
David Ingram, Professor of Health Informatics, CHIME, University College London

**Founding Members**
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.95</strong></td>
<td>T Beale, S Heard</td>
<td>14 Mar 2005</td>
</tr>
<tr>
<td>0.6</td>
<td>CR-000127. Restructure archetype specifications. Minor text changes.</td>
<td>T Beale, S Heard</td>
<td>14 Mar 2005</td>
</tr>
<tr>
<td></td>
<td><strong>RELEASE 0.9</strong></td>
<td>T Beale, S Heard</td>
<td>28 Dec 2003</td>
</tr>
<tr>
<td>0.5</td>
<td>Initial Writing. Based on Material taken from “A Shared Archetype and Template Language, Part I: A Position Paper for HL7, CEN TC 251, openEHR and other organisations”.</td>
<td>T Beale, S Heard</td>
<td>28 Dec 2003</td>
</tr>
</tbody>
</table>
# Table of Contents

1 **Introduction** ........................................................................................................ 7  
1.1 Purpose.................................................................................................................. 7  
1.2 Related Documents ............................................................................................... 7  
1.3 Status........................................................................................................................ 7  
2 **Definitions** ........................................................................................................... 8  
3 **Purpose of Archetypes and Templates** ............................................................... 9  
3.1 Purpose of Archetypes ............................................................................................ 9  
3.2 Purpose of Templates .............................................................................................. 9  
4 **Principles** .............................................................................................................. 10  
4.1 Overview.................................................................................................................. 10  
4.2 Formal Principles ................................................................................................... 10
1 Introduction

1.1 Purpose

This document describes a generic object model for archetypes, based only upon the generally accepted semantics of object models (typified by the OMG UML meta-model). The model presented here can be used as a basis for building software that processes archetypes, independent of their persistent representation; equally, it can be used to develop the output side of parsers that process archetypes in a linguistic format, such as the openEHR Archetype Definition Language (ADL) [4], XML-instance and so on. As a specification, it can be treated as an API for archetypes.

It is recommended that the openEHR ADL document [4] be read in conjunction with this document, since it contains a detailed explanation of the semantics of archetypes, and many of the examples are more obvious in ADL, regardless of whether ADL is actually used with the object model presented here or not.

1.2 Related Documents

Related documents include:

- The openEHR Archetype Definition Language (ADL)
- The openEHR Archetype Object Model (AOM)

1.3 Status

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

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

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

The definitions of the terms "archetype", "template" and variants as used in this paper are as follows:

archetype: a computable expression of a domain content model in the form of structured constraint statements, based on some reference model. openEHR archetypes are based on the openEHR reference model. Archetypes are all expressed in the same formalism. In general, they are defined for wide re-use, however, they can be specialized to include local particularities. They can accommodate any number of natural languages and terminologies.

template: a directly, locally usable definition which composes archetypes into a larger structure logically corresponding to a screen form. A templates may add further local constraints on the archetypes it mentions, including removing or mandating optional sections, and may define default values.
3 Purpose of Archetypes and Templates

3.1 Purpose of Archetypes
Archetypes are created for a number of purposes (described in detail in [1] and [3]), summarised here:

*Human Communication*: to enable domain concepts to be modelled in a formal way by domain experts;

*Specialised Searching*: also to compare data to specialised archetypes, or "predicates".

Archetypes can be used directly for the computational purposes described below, but are normally encapsulated by templates for this purpose. The key benefits of archetypes include:

*Knowledge-enabled systems*: the separation of information and knowledge concerns in software systems, allowing cheap, future-proof software to be built;

*Knowledge-level interoperability*: the ability of systems to reliably communicate with each other at the level of knowledge concepts;

*Domain empowerment*: the empowerment of domain specialists to define the informational concepts they work with, and have direct control over their information systems.

*Intelligent Querying*: to be used at runtime to enable the efficient querying of data based on the structure of archetypes from which the data was created.

3.2 Purpose of Templates
Templates constitute a form of constraint statement model, which is directly usable for:

*Data Construction*: to be used at runtime to constrain the creation of data in local contexts to conform to data capture requirements;

*Data Validation*: to be used at runtime to validate data from other sources.

While archetypes are generally broad models, and have very open compositional possibilities, templates are used to narrow the choices of archetypes for local or specific purposes. They can be used to control the following things:

- archetype composition, or *chaining*
- reduction in allowed terms
- restricting optionality
- removing structures defined in the referenced archetypes.
4 Principles

4.1 Overview

In this section we describe the principles of archetypes. Examples of domain-level concepts include "blood pressure", "physical examination (headings)", "biochemistry results" and so on. Here, the term reference model refers to any model which can have data instances in a computational system. The following figure illustrates the relationships of archetypes with data.

In this figure, the following relationships hold:

- data are instances of a reference model, such as an model of the EHR, Demographics or other concepts
- archetypes are instances of an "archetype model" which is a common formalism for expressing all archetypes
- the archetype model is formally related to the reference model, such that its semantics are those of constraint on objects of types defined in the reference model. It may also include linguistic elements allowing relationships between elements and invariants to be expressed (e.g. relationships between BMI and height and weight, apgar score and its 5 inputs etc)
- if data are created and modified using archetypes, archetypes constrain the configuration of data instances to be valid according to the archetype. E.g. Section and Entry objects are forced into a structure which is agreed to be correct for an ante-natal examination.

4.2 Formal Principles

These concepts can be stated in more formal terms as the following principles:

**Principle 1:** An archetype defines a whole, distinct, domain-level concept.

Archetypes must define coherent, whole concepts from the domain, in order to be useful. Archetypes enable distinct concepts to be recognised regardless of context. Accordingly,
there may be an archetype for "ECG result" since this is understood and used as a whole concept by clinicians, but not "ECG lead 2 result", which would only ever be understood as part of an "ECG result". The heart rate, as determined in an ECG, may be archetyped separately as this is a distinct concept that can be understood outside the current context. Similarly, we would not consider the heading "systolic" to be a meaningful archetype on its own, while it could be part of a "blood pressure" or "intravascular pressure" archetype.

**Principle 2:** an archetype defines constraints on reference model instances which express valid structure (i.e. composition, cardinality).

An archetype may be used to describe the general structuring of data instances to form a logical instance of a domain concept. For example, the hierarchical structure of headings used in problem-oriented recording would be visible in an archetype. In particular, this principle means that archetypes are not reliant on domain-specific semantics in the reference model (indeed, this is the whole point of archetypes - to avoid building such semantics into the reference model).

**Principle 3:** an archetype defines constraints on instances of a reference model which express valid types and values.

Archetypes also express constraints on allowable constructions of reference model instances, e.g., on allowed types, ordering, cardinality, values and so on. The combination of structure and constraint expression means that numerous variations on a data instance may conform to a single archetype.

**Principle 4:** the granularity of an archetype corresponds to the granularity of a business concept in an information model.

Archetypes are meaningfully defined at the same level of granularity as the "business" entities in the reference model. For example, if a reference model includes the business concepts "section headings" (a model of recursive headings in a document) and "entries" (a model of data taken from observations etc), then there will be archetypes for section heading trees, and for entries.

**Principle 5:** since each business concept (in the reference model) describes a particular ontological level found in the domain, all archetypes belong to one or other ontological level.

We can think of the archetypes at each of the ontological levels "thematic" (Documents, Composition), "organisational" (Sections, Organisers), "descriptive" (Entries) and so on as describing these levels of the domain. If there were 50 GP heading level archetypes, we could say that the organisational ontological level (from the point of view of GPs) was described by the archetypes.

**Principle 6:** a compositional relationship can exist between archetypes.

Archetypes can be composed to express valid possibilities for larger structures of data from different levels of the ontological hierarchy. For example, Section and Entry archetypes can be linked in a compositional way to define valid structures for the headings and data of "physical examination".

**Principle 7:** an archetype can be a specialisation of another archetype.

Archetypes can be defined at higher or lower levels of detail at a given ontological level. Thus, a "biochemistry results" archetype would define the general shape and constraints for all biochemistry results, while a "cholesterol results" archetype could be defined as a
specialisation of this, in order to further constrain data to conform only to the shape of a cholesterol test.

**Principle 8:** Archetypes are internally hierarchical in structure; that is to say, the constraints in an archetype has an internal hierarchical compositional structure.

This is because object models give rise to data that is inherently hierarchical in structure.

**Principle 9:** Archetype nodes, including the root and all leaves, are identified by node ids, which also also act as the codes for human-readable “meanings”.

Archetype node identifiers are defined as codes for terms local to the archetype. The definition of any such code acts as a standardised "design-time meaning", to distinguish it from an arbitrary "runtime name" which might be used in the data. Any node in an archetype can be referenced by concatenating the node identifiers from the archetype root to the node, to form an archetype path, which can be thought of as a “design-time” path.

**Principle 10:** Data generated from an archetype will have a compositional structure, in which nodes must be uniquely named, in order to be able to refer to them.

Since there can be repetitions of archetype structures in real data, the runtime name of a node is distinguished from the archetype name of the same node. Put another way, each node in runtime data has a name (due to the application or user at runtime) and a meaning (from the archetype). Additionally, each node from any given root point in data can be identified by its runtime path, formed by concatenating the runtime node names. An example might be multiple blood pressures taken over a period of time but each entered as single instances might be called 'resting blood pressure', 'standing blood pressure' and 'resting blood pressure @ 5 minutes'.

**Principle 11:** Different languages are dealt with via the usual means of translations through coded terminologies - this enables both archetypes at design time and data at runtime to appear totally in the local user's language.
The Archetype Definition Language (ADL)

Editors: {T Beale, S Heard}

Revision: 1.3.1

Pages: 105

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

---

1. Ocean Informatics Australia

© 2003, 2004 The openEHR Foundation

The openEHR foundation

is an independent, non-profit community, facilitating the creation and sharing
of health records by consumers and clinicians via open-source, standards-

Based implementations.

Founding Chairman

David Ingram, Professor of Health Informatics, CHIME, University

College London

Founding Members

Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

Patrons

To Be Announced

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the
   world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making
   presentations and education, so long as such purposes are non-commercial and
   are designed to comment on, further the goals of, or inform third parties
   about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you
   use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form:
   "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on
   a non-commercial basis. Accordingly, to the fullest extent permitted under
   applicable law, the openEHR Foundation accepts no liability and offers no
   warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy
   the materials and documents on this site other than as provided for in
   paragraphs 1 to 6 above, you must comply with the terms and conditions of the
   openEHR Free Commercial Use Licence, or enter into a separate written agreement
   with openEHR Foundation covering such activities. The terms and conditions of
   the openEHR Free Commercial Use Licence can be found at
   http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.9</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>CR-000079. Change interval syntax in ADL.</td>
<td>T Beale</td>
<td>24 Jan 2004</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
</tbody>
</table>
| 1.0   | CR-000077. Add cADL date/time pattern constraints.  
        CR-000078. Add predefined clinical types.  
        Better explanation of cardinality, occurrences and existence. | S Heard, T Beale | 14 Jan 2004 |
| 0.9.9 | CR-000073. Allow lists of Reals and Integers in cADL.  
        CR-000075. Add predefined clinical types library to ADL.  
        Added cADL and dADL object models. | T Beale, S Heard | 28 Dec 2003 |
| 0.9.8 | CR-000070. Create Archetype System Description.  
        Moved Archetype Identification Section to new Archetype System document.  
        Copyright Assigned by Ocean Informatics P/L Australia to The openEHR 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

“Microsoft” and “.Net” are registered trademarks of the Microsoft Corporation.

“Java” is a registered trademark of Sun Microsystems.

“Linux” is a registered trademark of Linus Torvalds.

Acknowledgements

Sebastian Garde, Central Qld University, Australia, for German translations.
The Archetype Definition Language (ADL)

Table of Contents

1 Introduction.............................................................................11
  1.1 Purpose.................................................................................................11
  1.2 Related Documents ..............................................................................11
  1.3 Nomenclature.......................................................................................11
  1.4 Status....................................................................................................11
  1.5 Peer review ..........................................................................................11
  1.6 Conformance........................................................................................12

2 Overview................................................................................. 13
  2.1 What is ADL? ......................................................................................13
    2.1.1 Structure.........................................................................................13
    2.1.2 An Example ...................................................................................14
    2.1.3 Semantics.......................................................................................15
  2.2 Computational Context ........................................................................16
  2.3 XML form of Archetypes ....................................................................16
  2.4 Changes From Previous Versions ........................................................17
    2.4.1 Version 1.3 from Version 1.2 .........................................................17
    2.4.2 Version 1.2 from Version 1.1 .........................................................17
    2.4.3 The Future: ADL Version 2.0 ........................................................19
  2.5 Tools.....................................................................................................19

3 dADL - Data ADL.................................................................. 21
  3.1 Overview..............................................................................................21
  3.2 Basics ...................................................................................................22
    3.2.1 Scope of a dADL Document..........................................................22
    3.2.2 Keywords.......................................................................................22
    3.2.3 Comments ......................................................................................22
    3.2.4 Quoting ..........................................................................................22
    3.2.5 Information Model Identifiers .......................................................22
    3.2.6 Instance Node Identifiers...............................................................23
    3.2.7 Semi-colons ...................................................................................23
  3.3 Paths.....................................................................................................23
  3.4 Structure...............................................................................................23
    3.4.1 General Form.................................................................................23
    3.4.1.1 Outer Delimiters ..................................................................................24
    3.4.1.2 Paths ............................................................................................................24
    3.4.2 Empty Sections ..............................................................................25
    3.4.3 Container Objects ..........................................................................25
    3.4.3.1 Paths ............................................................................................................27
    3.4.4 Nested Container Objects ..............................................................27
    3.4.4.1 Paths ............................................................................................................27
    3.4.5 Adding Type Information ..............................................................28
    3.4.6 Associations and Shared Objects...................................................28
    3.4.6.1 Paths ............................................................................................................29
  3.5 Leaf Data..............................................................................................30
    3.5.1 Primitive Types..............................................................................30
    3.5.1.1 Character Data .............................................................................................30
    3.5.1.2 String Data ..................................................................................................30
    3.5.1.3 Integer Data .................................................................................................30
The Archetype Definition Language (ADL)

3.5.1.4 Real Data ................................................................. 30
3.5.1.5 Boolean Data .......................................................... 31
3.5.1.6 Dates and Times ..................................................... 31
3.5.1.7 URIs ........................................................................... 32
3.5.2 Intervals of Ordered Primitive Types .................. 32
3.5.3 Lists of Primitive Types ........................................... 32
3.6 Data Navigation .............................................................. 32
3.7 Expression of dADL in XML .................................... 33
3.8 Syntax Specification ..................................................... 34
3.8.1 Grammar .................................................................. 34
3.8.2 Symbols ..................................................................... 40
3.9 Syntax Alternatives ....................................................... 41
3.9.1 Container Attributes .................................................. 42

4 cADL - Constraint ADL ................................................. 43
4.1 Overview ................................................................. 43
4.2 Basics .......................................................................... 44
4.2.1 Keywords .............................................................. 44
4.2.2 Comments ............................................................... 45
4.2.3 Information Model Identifiers ................................ 45
4.2.4 Node Identifiers ....................................................... 45
4.2.5 Natural Language .................................................... 45
4.3 Structure ...................................................................... 45
4.3.1 Complex Objects ..................................................... 46
4.3.2 Attribute Constraints ............................................... 47
4.3.2.1 Existence .......................................................... 47
4.3.3 Single-valued Attributes ......................................... 48
4.3.4 Container Attributes ............................................... 49
4.3.4.1 Cardinality ........................................................... 49
4.3.4.2 Occurrences ........................................................ 49
4.3.5 “Any” Constraints .................................................... 50
4.3.6 Object Node Identification and Paths ................... 51
4.3.7 Internal References ................................................. 53
4.3.8 Archetype Slots ....................................................... 53
4.3.9 Mixed Structures ..................................................... 55
4.4 Constraints on Primitive Types ................................. 55
4.4.1 Constraints on String ............................................... 56
4.4.1.1 List of Strings ......................................................... 56
4.4.2 Constraints on Integer .............................................. 57
4.4.2.1 List of Integers ....................................................... 57
4.4.2.2 Interval of Integer .................................................. 58
4.4.3 Constraints on Real .................................................. 58
4.4.4 Constraints on Boolean ............................................ 58
4.4.5 Constraints on Character ........................................ 58
4.4.5.1 List of Characters ................................................ 58
4.4.5.2 Regular Expression .............................................. 58
4.4.6 Constraints on Dates, Times and Durations ........... 59
4.4.6.1 Patterns .............................................................. 59
4.4.6.2 List and Intervals of Dates, Times, etc ..................... 60
4.4.7 Constraints on Lists of Primitive types ................. 60
The Archetype Definition Language (ADL)

4.4.8 Assumed Values........................................................................................................60
4.5 Syntax Specification.......................................................................................................61
4.5.1 Grammar....................................................................................................................61
4.5.2 Symbols.......................................................................................................................65
5 Assertions.........................................................................................................................71
5.1 Overview.........................................................................................................................71
5.2 Keywords..........................................................................................................................71
5.3 Operators........................................................................................................................71
5.3.1 Arithmetic Operators..................................................................................................72
5.3.2 Equality Operators......................................................................................................72
5.3.3 Relational Operators..................................................................................................72
5.3.4 Boolean Operators......................................................................................................72
5.3.5 Quantifiers..................................................................................................................72
5.4 Operands..........................................................................................................................73
5.5 Precedence and Parentheses..........................................................................................73
5.6 Future...............................................................................................................................73
5.6.1 Variables.......................................................................................................................73
6 ADL Paths.........................................................................................................................75
6.1 Overview..........................................................................................................................75
6.2 Relationship with W3C Xpath.........................................................................................75
6.3 Path Syntax.......................................................................................................................76
6.3.1 Grammar.......................................................................................................................76
6.3.2 Symbols.........................................................................................................................76
7 ADL - Archetype Definition Language..........................................................................79
7.1 Basics...............................................................................................................................79
7.1.1 Keywords.......................................................................................................................79
7.1.2 Node Identification.......................................................................................................79
7.1.3 Local Constraint Codes...............................................................................................80
7.2 Header Sections...............................................................................................................80
7.2.1 Archetype Section.........................................................................................................80
7.2.2 Controlled Indicator.....................................................................................................80
7.2.3 Specialise Section.........................................................................................................80
7.2.4 Concept Section...........................................................................................................80
7.2.5 Language Section and Language Translation...............................................................81
7.2.6 Description Section......................................................................................................81
7.3 Definition Section............................................................................................................82
7.4 Invariant Section..............................................................................................................83
7.5 Ontology Section..............................................................................................................84
7.5.1 Overview.......................................................................................................................84
7.5.2 Ontology Header Statements.......................................................................................85
7.5.3 Term_definition Section...............................................................................................85
7.5.4 Constraint_definition Section.......................................................................................86
7.5.5 Term_binding Section.................................................................................................86
7.5.6 Constraint_binding Section..........................................................................................87
7.6 Revision History Section..................................................................................................87
7.7 Validity Rules...................................................................................................................88
7.7.1 Global Archetype Validity............................................................................................88
7.7.2 Coded Term Validity ................................................................. 88
7.7.3 Definition Section ................................................................. 89
7.8 Syntax Specification ................................................................. 89
    7.8.1 Grammar ........................................................................... 89
    7.8.2 Symbols ............................................................................ 90
8 The ADL Parsing Process .............................................................. 92
    8.1 Overview ............................................................................... 92
9 Customising ADL ........................................................................ 95
    9.1 Introduction .......................................................................... 95
    9.1.1 Custom Syntax ................................................................. 95
    9.1.2 Custom Constraint Classes ................................................ 95
10 Relationship of ADL to Other Formalisms .................................. 99
    10.1 Overview ............................................................................ 99
    10.2 Constraint Syntaxes ............................................................. 99
    10.2.1 OCL (Object Constraint Language) .................................... 99
    10.3 Ontology Formalisms .......................................................... 99
    10.3.1 OWL (Web Ontology Language) ....................................... 99
    10.3.2 KIF (Knowledge Interchange Format) ................................ 101
    10.4 XML-based Formalisms ....................................................... 102
    10.4.1 XML-schema ................................................................. 102
1 Introduction

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

The intended audience includes:

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

1.2 Related Documents
Related documents include:

- The openEHR Archetype Object Model (AOM)
- The openEHR Archetype Profile (oAP)

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

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

The latest version of this document can be found in PDF format at http://svn.openehr.org/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
Introduction

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 artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Overview

2.1 What is ADL?

Archetype Definition Language (ADL) is a formal language for expressing archetypes, which are constraint-based models of domain entities, or what some might call “structured business rules”. The archetype concept is described by Beale [1], [2]. The openEHR Archetype Object Model [3] describes an object model equivalent of the ADL syntax. The openEHR archetype framework is described in terms of Archetype Definitions and Principles [6] and an Archetype System [7]. Other semantic formalisms which were considered in the course of archetype, and some which remain relevant are described in detailed in section 10 on page 99.

ADL uses three other syntaxes, cADL (constraint form of ADL), dADL (data definition form of ADL), and a version of first-order predicate logic (FOPL), to describe constraints on data which are instances of some information model (e.g. expressed in UML). It is most useful when very generic information models are used for describing the data in a system, for example, where the logical concepts PATIENT, DOCTOR and HOSPITAL might all be represented using a small number of classes such as PARTY and ADDRESS. In such cases, archetypes are used to constrain the valid structures of instances of these generic classes to represent the desired domain concepts. In this way future-proof information systems can be built - relatively simple information models and database schemas can be defined, and archetypes supply the semantic modelling, completely outside the software. ADL can thus be used to write archetypes for any domain where formal object model(s) exist which describe data instances.

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

2.1.1 Structure

Archetypes expressed in ADL resemble programming language files, and have a defined syntax. ADL itself is a very simple “glue” syntax, which uses two other syntaxes for expressing structured constraints and data, respectively. The cADL syntax is used to express the archetype definition, while the dADL syntax is used to express data which appears in the language, description, ontology, and revision history sections of an ADL archetype. The top-level structure of an ADL archetype is shown in FIGURE 1.
This main part of this document describes dADL, cADL and ADL path syntax, before going on to describe the combined ADL syntax, archetypes and domain-specific type libraries.

2.1.2 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=1.2)
  archetype_id
    [specialise]
      archetype_id
    concept
      concept_id
    language
      dADL:language details
        dADL: descriptive meta-data
        [declarations]
          FOQL: declaration statements
        definition
          cADL: formal constraint definition
        [invariant]
          FOQL: assertion statements
      ontology
        dADL: terminology and language definitions
      [revision_history]
        dADL: history of change audits

archetype (adl_version=1.2)
  adl-test-instrument.guitar.draft
concept
  [at0000] -- guitar
language
  original_language = <"en">
  translations = <"de">...

FIGURE 1 ADL Archetype Structure
```
2.1.3 Semantics

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

While ADL syntax remains the primary abstract formalism for expressing archetypes, the AOM defines the semantics of an archetype, in particular relationships which must hold true between the parts of an archetype for it to be valid as a whole.
2.2 Computational Context

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

Archetypes are applied to data via the use of templates, which are defined at a local level. Templates generally correspond closely to screen forms, and may be re-usable at a local or regional level. Templates do not introduce any new semantics to archetypes, they simply specify the use of particular archetypes, and default data values.

A third artifact which governs the functioning of archetypes and templates at runtime is a local palette, which specifies which natural language(s) and terminologies are in use in the locale. The use of a palette removes irrelevant languages and terminology bindings from archetypes, retaining only those relevant to actual use. FIGURE 3 illustrates the overall environment in which archetypes, templates, and a locale palette exist.

2.3 XML form of Archetypes

With ADL parsing tools it is possible to convert ADL to any number of forms, including various XML formats. XML instance can be generated from the object form of an archetype in memory. An XML-schema corresponding to the ADL Object Model will be published on openEHR.org.

FIGURE 2 Relationship of ADL with Object Models

ADL Language Definition (EBNF)

FIGURE 2 Relationship of ADL with Object Models
2.4 Changes From Previous Versions

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

2.4.1 Version 1.3 from Version 1.2

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.

2.4.2 Version 1.2 from Version 1.1

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

    archetype (adl_version=1.2)

It is strongly recommended that all tool implementors include this information when archetypes are saved, enabling archetypes to gradually become imprinted with their correct version, for more reliab-
ble 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 archetypes may be stored. This section is included at the end of the archetype because it is in general a monotonically growing section.
The Archetype Definition Language (ADL)

Overview
Rev 1.3.1

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.

2.4.3

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 1.2 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”] = <
author = <"edward.jones@translators.co.uk">
accreditation = <"British Medical Translator id 00400595">
>
[“ru”] = <
author = <"eva.sakharova@translators.co.ru">
accreditation = <"Russion Translator id 892230A">
>
>

2.5

Tools

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

Editors:{T Beale, S Heard}

Page 19 of 105
© 2003, 2004 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org

Date of Issue: 18 Jan 2006


3  dADL - Data ADL

3.1  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 = <
            forenames = "Sherlock"
            family_name = "Holmes"
            salutation = "Mr"
        >
        address = <
            habitation_number = "221B"
            street_name = "Baker St"
            city = "London"
            country = "England"
        >
    >

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

In the above the identifiers PERSON, name, address etc are all assumed to come from an information model. The basic design principle of dADL is to be able to represent data in a way that is both machine-processible and human readable, while making the fewest assumptions possible about the information model to which the data conforms. To this end, type names are only used (optionally) for root nodes; otherwise, only attribute names and values are explicitly shown; no syntactical assumptions are made about whether the underlying model is relational, object-oriented or what it actually looks like. More than one information model can be compatible 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, documented in this section, all other more sophisticated types are expressed structurally, with leaf values of these primitive types. Other domain-specific literal types are documented in Customising ADL on page 95.

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

- providing a more comprehensive set of leaf data types, including intervals of numerics and date/time types, and lists of all primitive types;
- adhering to object-oriented semantics, which XML schema languages generally do not;
not using the confusing XML notion of ‘attributes’ and ‘elements’ to represent what are essentially object properties.

Of course, this does not prevent XML exchange syntaxes being used for dADL, and indeed the conversion to XML instance is rather straightforward. Details on the XML expression of dADL and use of Xpath expressions is described in section 3.6 on page 32 and section 3.7 on page 33.

3.2 Basics

3.2.1 Scope of a dADL Document
A dADL document may contain one or more objects from the same object model.

3.2.2 Keywords

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

3.2.3 Comments

In a dADL text, comments satisfy the following rule:

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

3.2.4 Quoting

The dADL quoting rule is as follows:

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

3.2.5 Information Model Identifiers

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

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

Type names in this document are in all uppercase, e.g. PERSON, 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 types assumed from constructed types.

Attribute names are shown in all lowercase, e.g. home_address. In both cases, underscores are used to represent word breaks. This convention is used to maximise the readability of this document, and other conventions may be used, such as the common programmer’s mixed-case convention exemplified by Person and homeAddress, as long as they obey the rule above. The convention chosen for any particular dADL document should be based on the convention used in the underlying information model. Identifiers are shown in green in this document.
3.2.6 Instance Node Identifiers

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

Instance nodes can be identified in dADL using an identifier delimited by brackets, e.g. [some_id]. Any dADL identifier may appear within the brackets, depending on how it is used.

Instance identifiers may be used to identify and refer to data expressed in dADL, but also to external entities. Instance identifiers are shown in this document in magenta.

3.2.7 Semi-colons

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

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

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

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

Semi-colons are completely optional.

3.3 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 6 on page 75. A path either finishes in a slash, and identifies an object node, or finishes in an attribute name, and identifies an attribute node. A typical ADL path used to refer to a node in a dADL text is as follows.

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

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

3.4 Structure

3.4.1 General Form

A dADL document or text is always a hierarchical expression of one or more instances of one or more object or entity types. In its simplest form, a dADL text consists of repetitions of the following pattern:

```
attribute_id = <value>
```

In the most basic form of dADL, each attribute id is the name of an attribute in an implied or real object or relational model of data. Each “value” is either a literal value of a primitive type (see Primitive Types on page 30) or a further nesting of attribute names and values, terminating in leaf nodes of
primitive type values. Where sibling attribute nodes occur, the attribute identifiers must be unique, just as in a standard object or relational model.

**Sibling attribute identifiers must be unique.**

The following shows a typical structure.

```plaintext
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 3.4.6 on page 28.

3.4.1.1 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:

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

**Outer ‘<>’ delimiters in a dADL text are optional.**

3.4.1.2 Paths

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

```plaintext
aaa
aaa/bbbbb
aaa/bbbbb/xx/ -- path to a leaf value
aaa/ccc
aaa/ccc/pppp/
aaa/ccc/pppp/xx -- path to a leaf value
aaa/ccc/gg/
ddd -- path to a leaf value
```
3.4.2 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.**

3.4.3 Container Objects

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

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

See Lists of Primitive Types on page 32 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 **qualifiers**. No attribute name is explicitly given; any primitive comparable value is allowed as the qualifier, 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 qualifiers [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. The following example shows the use of string values as qualifiers 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 qualifiers follows exactly the same syntax described below for data of primitive types. It is convenient in some cases to construct qualifier 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.

```plaintext
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”>

  >

  subjects = <
    [“philosophy:plato”] = < -- note construction of qualifier
      name = <“philosophy”>
      teacher = <”plato”>
      topics = <"meta-physics", “natural science”>
      weighting = <76%>

    [“philosophy:kant”] = <
      name = <“philosophy”>
      teacher = <”kant”>
      topics = <“meaning and reason”, “meta-physics”, “ethics”>
      weighting = <80%>

    >

    [“art”] = <
      name = <“art”>
      teacher = <”goya”>
      topics = <“technique”, “portraiture”, “satire”>
      weighting = <78%

    >

>
```

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

The example above conforms directly to an object-oriented type specification of the following form:

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

© 2003, 2004 The openEHR Foundation

email: info@openEHR.org web: http://www.openEHR.org
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 qualifier 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).

3.4.3.1 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”

3.4.4 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”>
    >
>
```

3.4.4.1 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]
```
3.4.5 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 simply by including the type name after the ‘=’ sign, as in the following example.

```dADL
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.

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

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

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

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

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

3.4.6 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 association. This can be expressed as follows.

```xml
destinations = <
  ["seville"] = <
    hotels = <
      ["gran sevilla"] = </hotels["gran sevilla"]>
      ["sofitel"] = </hotels["sofitel"]>
      ["hotel real"] = </hotels["hotel real"]>
    >
  >
>
bookings = <
  ["seville:0134"] = <
    customer_id = <"0134">
    period = <...>
    hotel = </hotels["sofitel"]>
  >
>
hotels = <
  ["gran sevilla"] = HISTORIC_HOTEL <>
  ["sofitel"] = LUXURY_HOTEL <>
  ["hotel real"] = PENSION <>
>
```

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

**Shared objects are referenced using paths.**

### 3.4.6.1 Paths

The path set from the above example is as follows:

```xml
/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"]/
```
3.5 Leaf Data

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

\[ \text{some_attribute} = \langle \text{manifest value here} \rangle \]

3.5.1 Primitive Types

3.5.1.1 Character Data

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

'\text{a}'

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

'&ohgr;' -- greek omega

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

3.5.1.2 String Data

All strings are enclosed in double quotes, as follows:

"\text{this is a string}"  

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

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

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

"\text{a &isin; A}" -- prints as: \text{a ∈ A}

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

3.5.1.3 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 3.5.3 below).

3.5.1.4 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 3.5.3 below).

### 3.5.1.5 Boolean Data

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

- **True**
- **False**

### 3.5.1.6 Dates and Times

In dADL, full and partial dates, times and durations can be expressed. All full dates, times and durations are expressed in ISO8601 form. Patterns for dates and times based on ISO8601 include the following:

- **yyyy-MM-dd** -- a date
- **hh:mm:ss[.sss][Z]** -- a time with optional seconds
- **yyyy-MM-dd hh:mm:ss[.sss][Z]** -- a date/time

where:

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

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

- **1919-01-23** -- birthdate of Django Reinhardt
- **16:35:04.5** -- rise of Venus in Sydney on 24 Jul 2003
- **2001-05-12 07:35:20+1000** -- timestamp on an email received from Australia
- **P22D4H15M0S** -- period of 22 days, 4 hours, 15 minutes

Partial dates and times, i.e. dates and times with unknown parts are expressed in the same form but with the literal "??" for the unknown parts. **This is a deviation from ISO8601**, which specifies that partial dates and times can be expressed simply by omitting the unknown parts. However, this leads to strings like “12” (12 o’clock, minutes and seconds unknown), which cannot easily be recognised by parsers. Instead, dADL uses forms like “12:??:??” which are easily parsed as being dates and times. 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-dd hh:mm:??** -- date/time with unknown seconds
- **yyyy-MM-dd hh:??:??** -- date/time with unknown minutes and seconds
- **yyyy-MM-dd ??:??:??** -- date/time with unknown time
- **yyyy-??-?? ??:??:??** -- date/time with unknown month, day and time
- **yyyy-??-?? ??:??:??** -- date/time with unknown month, day and time
3.5.1.7 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/rfc2396.txt](http://www.ietf.org/rfc/rfc2396.txt). In the context of dADL, no quotes or inverted commas are needed; and neither spaces nor angle brackets are allowed; both have to be quoted e.g. %20 means a space (as per the URI standard). Examples of URIs in dADL:

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

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

Examples of this syntax include:

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

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

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

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

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

- “en”, ... -- languages
- “icd10”, ... -- terminologies
- [at0200], ... -- terminologies

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

3.6 Data Navigation
Characteristics of the Syntax
The dADL syntax as described above has a number of useful characteristics which enable the extensive use of paths to navigate it, and express references. These include:
• each <> block corresponds to an object (i.e. an instance of some type in an information model);
• the name before an ‘=’ is always an attribute name or else a container element key, which attaches to the attribute of the enclosing block;
• paths can be formed by 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.

3.7 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 attribute nodes map to tagged nodes of the same name
• container attribute nodes map to a series of tagged nodes of the same name, each with the XML attribute ‘id’ set to the dADL key. For example, the dADL:

```dADL
subjects = <
  "philosophy:plato" = <
    name = "philosophy"
  >
  "philosophy:kant" = <
    name = "philosophy"
  >
>
```

maps to the XML:

```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 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:

```dADL
cities = <
  [spain] = <
```
maps to the XML in which the synthesised attribute tag “_items” appears:

```xml
<cities id="spain">
  <_items id="hotels">
    ...
  </_items>
  <_items id="attractions">
    ...
  </_items>
</cities>

<cities id="egypt">
  <_items id="hotels">
    ...
  </_items>
  <_items id="attractions">
    ...
  </_items>
</cities>
```

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

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

```dADL
destinations = <
  [“seville”] = TOURIST_DESTINATION <
    profile = DESTINATION_PROFILE <>
    hotels = <
      [“gran sevilla”] = HISTORIC_HOTEL <>
    >
  >
</destinations>
```

maps to:

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

### 3.8 Syntax Specification

#### 3.8.1 Grammar
This section shows the dADL grammar, as implemented and tested in the openEHR implementation project.
---
dADL grammar @changeset 1.32 (2004-11-15),
-- openEHR implem-dev BK repository
---

input: attr_vals
    | error
;
-------------- body ---------------

attr_vals: attr_val
    | attr_vals attr_val
    | attr_vals ';' attr_val
;
attr_val: attr_id SYM_EQ object_block -- could be a single or multiple attr
;
object_block: single_attr_object_block
    | multiple_attr_object_block
;

multiple_attr_object_block: untyped_multiple_attr_object_block
    | V_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
;

attr_id: V_ATTRIBUTE_IDENTIFIER
    | V_ATTRIBUTE_IDENTIFIER error
;

object_key: '][' simple_value ']
;

single_attr_object_block: untyped_single_attr_object_block
    | V_TYPE_IDENTIFIER untyped_single_attr_object_block
;

untyped_single_attr_object_block: single_attr_object_complex_head
empty_object_complex_body
    | single_attr_object_complex_head single_attr_object_complex_body
    | single_attr_object_primitive
;
single_attr_object_complex_head: SYM_START_DBLOCK
;

single_attr_object_complex_body: attr_vals SYM_END_DBLOCK
;

empty_object_complex_body: SYM_END_DBLOCK
;

single_attr_object_primitive: 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
                    | query
;

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

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

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

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

---------------------- BASIC DATA VALUES -----------------------
string_value: V_STRING
  {
  \$\$ := \$1
  }

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

integer_value: V_INTEGER
  | '+' V_INTEGER
  | '-' V_INTEGER

integer_list_value: integer_value ',' integer_value
  | integer_list_value ',' integer_value
  | integer_value ',' SYM_LIST_CONTINUE

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

© 2003, 2004 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
character_value: V_CHARACTER ;

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

date_value: precise_date_value
               | partial_date_value ;

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

partial_date_value: V_INTEGER '-' V_INTEGER '-' SYM_DT_UNKNOWN
               -- in ISO8601 form yyyy-MM-??
               | V_INTEGER '-' SYM_DT_UNKNOWN '-' SYM_DT_UNKNOWN
               -- in ISO8601 form yyyy-??-?? ;

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

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

time_value: precise_time_value
               | precise_time_value time_zone
               | partial_time_value
               | partial_time_value time_zone ;

precise_time_value: V_INTEGER ':' V_INTEGER ':' V_INTEGER
               | V_INTEGER ':' V_INTEGER ':' V_REAL
               | V_INTEGER ':' V_INTEGER ;

partial_time_value: V_INTEGER ':' V_INTEGER ':' SYM_DT_UNKNOWN
               | V_INTEGER ':' SYM_DT_UNKNOWN ':' SYM_DT_UNKNOWN ;

time_zone: 'Z'
               | '+' V_INTEGER ;

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

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

date_time_value: precise_date_time_value 
| partial_date_time_value 
| 

precise_date_time_value: precise_date_value precise_time_value 
| 

partial_date_time_value: precise_date_value partial_time_value 
| precise_date_value SYM_DT_UNKNOWN ':' SYM_DT_UNKNOWN ':' SYM_DT_UNKNOWN 
| partial_date_value SYM_DT_UNKNOWN ':' SYM_DT_UNKNOWN ':' SYM_DT_UNKNOWN 
| 

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 
| 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: duration_magnitude 
| '-' duration_magnitude 
| 

duration_magnitude: V_ISO8601_DURATION 
| 

duration_list_value: duration_value ',' duration_value 
| duration_list_value ',' duration_value 
| duration_value ',' SYM_LIST_CONTINUE 
| 

duration_interval_value: SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS 
| 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
3.8.2 Symbols
The following specifies the symbols and lexical patterns used in the above grammar.

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

/* symbols */

"-" Minus_code
"+" Plus_code
"*" Star_code
"/" Slash_code
"^" Caret_code
"." Dot_code
";" Semicolon_code
"," Comma_code
":" Colon_code
"!" Exclamation_code
"(" Left_parenthesis_code
")" Right_parenthesis_code
"$" Dollar_code
"??" SYM_DT_UNKNOWN
"?" Question_mark_code
"|" SYM_INTERVAL_DELIM
"[" Left_bracket_code
"]" Right_bracket_code
"=" SYM_EQ
">=" SYM_GE
"<=" SYM_LE
"<" SYM_LT / SYM_START_DBLOCK
">" SYM_GT / SYM_END_DBLOCK
".." SYM_ELLIPSIS
"..." SYM_LIST_CONTINUE
```

/* keywords */
3.9 Syntax Alternatives

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

A perfectly reasonable alternative to the syntax chosen in dADL 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 qualifier for each container member, giving syntax like the following:

```plaintext
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.:

```plaintext
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:

```plaintext
/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 omit “_items”, but retain other container pseudo-attributes, thus:

```plaintext
people = <
    [1] = <name = <> birth_date = <> interests = <>>
    [2] = <name = <> birth_date = <> interests = <>>
    [3] = <name = <> birth_date = <> interests = <>>
    _is_ordered = <True>
    _upper = <200>
>
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 appropriate container types from a recognised standard such as OMG UML, OCL or IDL.
4 cADL - Constraint ADL

4.1 Overview

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

```plaintext
PERSON[at0000] matches { -- constraint on PERSON instance
    name matches { -- constraint on PERSON.name
        TEXT matches {/.*/} -- any non-empty string
    }
    addresses cardinality matches {0..*} matches { -- constraint on
        ADDRESS matches { -- PERSON.addresses
            -- etc --
        }
    }
}
```

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

```plaintext
PERSON[at0000] ∈ { -- constraint on PERSON instance
    name ∈ { -- constraint on PERSON.name
        TEXT ∈ {/.*/} -- any non-empty string
    }
    addresses cardinality ∈ {0..*} ∈ { -- constraint on
        ADDRESS ∈ { -- PERSON.addresses
            -- etc --
        }
    }
}
```

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

In the standard cADL documented in this section, literal leaf values (such as the regular expression `/.*/` in the above example) are always constraints on a set of ‘standard’ widely-accepted primitive types, as described in the dADL section. Other more sophisticated constraint syntax types are described in cADL - Constraint ADL on page 43.
4.2 Basics

4.2.1 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
- include, exclude

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

<table>
<thead>
<tr>
<th>Textual Rendering</th>
<th>Symbolic Rendering</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches, is_in</td>
<td>∈</td>
<td>Set membership, “p is in P”</td>
</tr>
<tr>
<td>not, ~</td>
<td>∼</td>
<td>Negation, “not p”</td>
</tr>
</tbody>
</table>

Keywords are shown in blue in this document.

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

```
aaa matches {/.*ion[^\s\n\t]/} -- the set of english words ending in ‘ion’
```

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

```
XXX matches {
    aaa matches {
        "Yyy" matches {0..3} -- the value space of the
    } bbb matches {
        "zzz" matches {>1992-12-01} -- and instance of XXX
    }
}
```

The meaning of the constraint structure above is: in data matching the constraints, there is an instance of type XXX whose attribute values recursively match the inner constraints named after those attributes, and so on, to the leaf level.

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}
```

---

1. was ‘use_archetype’, which is now deprecated
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`.

### 4.2.2 Comments

In a cADL 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.
```

### 4.2.3 Information Model Identifiers

As with dADL, identifiers from the underlying information model are used for all cADL nodes. Identifiers obey the same rules as in dADL: type names commence with an upper case letter, while attribute and function names commence with a lower case letter. In cADL, type names and any property (i.e. attribute or function) name can be used, whereas in dADL, only type names and attribute names appear.

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

Type 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. Identifiers are shown in green in this document.

### 4.2.4 Node Identifiers

In cADL, an entity in brackets e.g. `[xxxx]` is used to identify “object nodes”, i.e. nodes expressing constraints on instances of some type. Object nodes always commence with a type name. Any string may appear within the brackets, depending on how it is used. However, in this document, all node identifiers are of the form of an archetype term identifier, i.e. `[atNNNN]`, e.g. `[at0042]`. Node identifiers are shown in magenta in this document.

### 4.2.5 Natural Language

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

### 4.3 Structure

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

```
PERSON[at0001] ∈ {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`. 
In the above, any identifier (shown in green) followed by the \( \in \) operator (equivalent text keyword: \textit{matches} or \textit{is in}) followed by an open brace, is the start of a “block”, which continues until the closing matching brace (normally visually indented to come under the start of the line at the beginning of the block).

The example above expresses a constraint on an instance of the type \texttt{PERSON}; the constraint is expressed by everything inside the \texttt{PERSON} block. The two blocks at the next level define constraints on properties of \texttt{PERSON}, in this case \texttt{name} and \texttt{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 properties (of that type), followed by types (being the types of the attribute under which it appears) until leaf nodes are reached.

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

![FIGURE 4 Object and Attribute Blocks in cADL](image)

4.3.1 Complex Objects

It may by now be clear that the identifiers in the above could correspond to entities in an object-oriented information model. A UML model compatible with the example above is shown in FIGURE 5. Note that there can easily be more than one model compatible with a given fragment of cADL syntax, and in particular, there may be more properties and classes in the reference model than are mentioned in the cADL constraints. In other words, a cADL text includes constraints \textit{only for those parts of a model which are useful or meaningful to constrain}.

Constraints expressed in cADL cannot be stronger than those from the information model. For example, the \texttt{PERSON.family\_name} attribute is mandatory in the model in FIGURE 5, so it is not valid to
express a constraint allowing the attribute to be optional. In general, a cADL archetype can only further constrain an existing information model. However, it must be remembered that for very generic models consisting of only a few classes and a lot of optionality, this rule is not so much a limitation as a way of adding meaning to information. Thus, for a demographic information model which has only the types PARTY and PERSON, one can write cADL which defines the concepts of entities such as COMPANY, EMPLOYEE, PROFESSIONAL, and so on, in terms of constraints on the types available in the information model.

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

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

### 4.3.2 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>.

#### 4.3.2.1 Existence

The only constraint that applies to all attributes is to do with existence. Existence constraints say whether an attribute must exist, and are indicated by “0..1” or “1” markers at line ends in UML diagrams (and often mistakenly referred to as a “cardinality of 1..1”). It is the absence or presence of the cardinality constraint in cADL which indicates that the attribute being constrained is single-valued or a container attribute, respectively. Existence constraints are expressed in cADL as follows:

```cADL
QUANTITY matches {
  units existence matches (0..1) matches {"mm[Hg]"}
}
```
The meaning of an existence constraint is to indicate whether a value - i.e. an object - is mandatory or optional (i.e. obligatory or not) in runtime data for the attribute in question. The above example indicates that a value for the ‘units’ attribute is optional. The same logic applies whether the attribute is of single or multiple cardinality, i.e. whether it is a container or not. For container attributes, the existence constraint indicates whether the whole container (usually a list or set) is mandatory or not; a further cardinality constraint (described below) indicates how many members in the container are allowed.

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

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

### 4.3.3 Single-valued Attributes

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

```plaintext
ELEMENT[at0004] matches {                      -- speed limit
  value matches {
    QUANTITY matches {
      magnitude matches {0..55}
      property matches ("velocity")
      units matches {"mph"}      -- miles per hour
    }
    QUANTITY matches {
      magnitude matches {0..100}
      property matches ("velocity")
      units matches {"km/h"}      -- km per hour
    }
  }
}
```

Here, the cardinality of the `value` attribute is 1..1 (the default), while the occurrences of both `QUANTITY` constraints is optional, leading to the result that only one `QUANTITY` instance can appear in runtime data, and it can match either of the constraints.

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

Note that there is a more efficient way to express the above example, using domain type extensions. An example is provided in section 10.1.2 on page 99.
4.3.4 Container Attributes

4.3.4.1 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:

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

```plaintext
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 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, what number of member items it must 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.

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:

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

4.3.4.2 Occurrences

A constraint on occurrences is used only with cADL object nodes (not attribute nodes), to indicate how many times in runtime data an instance of a given class conforming to a particular constraint can occur. It only has significance for objects which are children of a container attribute, since by definition, the occurrences of an object which is the value of a single-valued attribute can only be 0..1 or 1..1, and this is already defined by the attribute existence. However, it is not illegal. In the example below, three EVENT constraints are shown; the first one (“1 minute sample”) is shown as mandatory, while the other two are optional.

```plaintext
events cardinality ∈ \{\} ∈ {
  EVENT[at0002] occurrences ∈ \{1..1\} ∈ \{\} -- 1 min sample
```

Editors:[T Beale, S Heard]  Page 49 of 105  Date of Issue: 18 Jan 2006

© 2003, 2004 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
Another contrived example below expresses a constraint on instances of GROUP such that for GROUPs representing tribes, clubs and families, there can only be one “head”, but there may be many members.

GROUP[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 class may appear in data, each conforming to the one constraint block in the archetype. A single positive integer, or the infinity indicator, may also be used on its own, thus: {2}, {*}. The default occurrences, if none is mentioned, is {1..1}.

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

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

VCOC: cardinality/occurrences validity: the interval represented by: the sum of all occurrences minimum values - the sum of all occurrences maximum values must be inside the interval of the cardinality.

4.3.5 “Any” Constraints

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

PERSON[at0001] matches {
  name existence matches {0..1} matches {*}
  -- etc --
}

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

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

ELEMENT[at0004] matches {
  -- speed limit
The Archetype Definition Language (ADL)

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.

4.3.6 Object Node Identification and Paths

In many of the examples above, some of the object node typenames are followed by a node identifier, shown in brackets.

**Node identifiers** are required for any object node which is intended to be addressable elsewhere in the cADL text, or in the runtime system and which would otherwise be ambiguous i.e. has sibling nodes.

In the following example, the `PERSON` type does not require an identifier, since no sibling node exists at the same level, and unambiguous paths can be formed:

```cadl
members cardinality ∈ {*} ∈ {
    PERSON ∈ {
        title ∈ {"head"}
    }
}
```

The path to the `title` attribute is

```cadl
members/title
```

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

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

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

```cadl
members[at0104]/title
members[at0105]/title
```

Logically, all non-unique parent nodes of an identified node must also be identified back to the root node. The primary function of node identifiers is in forming paths, enabling cADL nodes to be unambiguously referred to. The node identifier can also perform a second function, that of giving a design-time meaning to the node, by equating the node identifier to some description. Thus, in the example shown in section 4.3.1, the `ELEMENT` node is identified by the code `[at0010]`, which can be designated elsewhere in an archetype as meaning “diastolic blood pressure”.

Node ids are required only where it is necessary to create paths, for example in “use” statements. However, the underlying reference model might have stronger requirements. The openEHR EHR information models [17] for example require that all node types which inherit from the class `LOCATA-`
BLE have both a `archetype_node_id` and a runtime `name` attribute. Only data types (such as `QUANTITY`, `CODED_TEXT`) and their constituent types are exempt.

Paths are used in cADL to refer to cADL nodes, and are expressed in the ADL path syntax, described in detail in section 7 on page 81. ADL paths have the same alternating object/attribute structure implied in the general hierarchical structure of cADL, obeying the pattern `TYPE/attribute/TYPE/attribute/...`.

**Paths in cADL always refer to object nodes, and can only be constructed through nodes having node ids, or nodes which are the only child object of a single-cardinality attribute.**

Unusually for a path syntax, a trailing object identifier can be required, even if the attribute corresponds to a single relationship (as might be expected with the “name” property of an object) because in cADL, it is legal to define multiple alternative object constraints - each identified by a unique node id - for a relationship node which has single cardinality.

Consider the following cADL example:

```
HISTORY occurrences ∈ {1} ∈ {
    periodic ∈ {False}
    events cardinality ∈ {1} ∈ {
        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
```

It is valid to add attribute references to the end of a path, if the underlying information model permits it, as in the following example.

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

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

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

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

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

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

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

An archetype internal reference is introduced with the use_node keyword, in a line of the following form:

use_node TYPE object_path

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

PERSON ∈ {  
  identities ∈ {  
    -- etc --  
  }  
  contacts cardinality ∈ {0..*} ∈ {  
    CONTACT [at0002] ∈ {  
      purpose ∈ {-- etc --}  
      addresses ∈ {-- etc --}  
    }  
    CONTACT [at0003] ∈ {  
      purpose ∈ {-- etc --}  
      addresses ∈ {-- etc --}  
    }  
    CONTACT [at0004] ∈ {  
      purpose ∈ {-- etc --}  
      addresses cardinality ∈ {0..*} ∈ {  
        ADDRESS [at0005] ∈ {  
          type ∈ {-- etc --}  
          details ∈ {-- etc --}  
        }  
        ADDRESS [at0006] ∈ {  
          type ∈ {-- etc --}  
          details ∈ {-- etc --}  
        }  
        ADDRESS [at0007] ∈ {  
          type ∈ {-- etc --}  
          details ∈ {-- etc --}  
        }  
      }  
    }  
    CONTACT [at0008] ∈ {  
      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  
      }  
    }  
  }  
}

4.3.8 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”, 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 75). The slot might be “wide”, meaning it allows numerous other archetypes, or “narrow”, where it allows only a few or just one archetype. The point at where the slot occurs in the archetype is a “chaining point”, i.e. a point at which archetypes are chained together. An archetype slot is defined in terms of two lists of assertions statements defining which archetypes are allowed to and excluded from filling that slot. Since archetype slots are typed, the (possibly abstract) type of the allowed archetypes is already constrained. Otherwise, any assertion can be made. A basic kind of assertion is on the id of archetypes allowed in the slot, and it is possible to limit this to a single archetype, meaning that the choice of archetype in that slot is fixed. In general, however, the intention of archetypes is to provide highly re-usable models of real world content with local constraining left to templates. The full semantics of archetype chaining are described in the “Archetype Object Model” document.

An archetype slot is introduced with the keyword allow_archetype, and is expressed using two lists of assertions, each introduced with the keywords include and exclude, respectively.

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

```plaintext
SECTION [at2000] occurrences ∈ {0..1} ∈ { -- objective
  items ∈ {
    allow_archetype ENTRY occurrences ∈ {0..1} ∈ {
      include
      concept_short_name ∈ {/.+/}
    }
    allow_archetype SECTION occurrences ∈ {0..*} ∈ {
      include
      id ∈ {/.*\.iso-ehr\.section\..*\..*/}
      exclude
      id ∈ {/.*\.iso-ehr\.section\..patient_details\..*/}
    }
  }
}
```

Here, every constraint inside the block starting on an allow_archetype line contains constraints that must be met by archetypes in order to fill the slot. (Note that in the examples above, and in all ADL, the ‘\’ character in paths is not a path separator, as some Microsoft Windows users might assume - it retains its meaning as a quoting character; above it is used to quote the ‘.’ character to ensure it has its literal meaning of ‘.’ rather than “any character” which is its regular expression meaning. Although not particularly beautiful, the syntax used above to indicate allowed values of the id attribute is completely standard regular expression syntax, and would be familiar to most users of Unix operating systems (e.g. Linux, BSD etc), Perl, and many other formalisms. Developers of archetypes using GUI tools should of course be spared such technical details.)

Other constraints are possible as well, including that the allowed archetype must contain a certain keyword, or a certain path. The latter is quite powerful – it allows archetypes to be linked together on the basis of context. For example, under a “genetic relatives” heading in a Family History Organiser archetype, the following logical constraint might be used:

```plaintext
allow_archetype ENTRY occurrences ∈ {0..*} matches {
  include
  short_concept_name ∈ {“family_history_subject”}
  ∧ ∃ /subject/.relation
      → /subject/.relation ∈ {
          CODED_TEXT ∈ {
```
4.3.9 Mixed Structures

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

- **complex object structures**: any node introduced by a type name and followed by {} containing constraints on attributes;
- **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.

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

```cADL
SECTION[at2000] ∈ {  
    items_cardinality ∈ {0..*; ordered} ∈ {  
        ENTRY[at2001] ∈ {-- etc --}  
        allow_archetype ENTRY ∈ {-- etc --}  
        use_node ENTRY [at0001]/some_path[at0004]/ENTRY[at2002] ∈ {-- etc --}  
        use_node ENTRY [/at1002]/some_path[at1012]/ENTRY[at1005]/some_path[at1052]/ENTRY[at2003] ∈ {-- etc --}  
    }  
}  
```

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

4.4 Constraints on Primitive Types

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

```cADL
some_attr matches {some_pattern}  
```

rather than:

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

This is made possible because the syntax patterns of all primitive types 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. cADL does not however obligate the brief form described here, and the more verbose one can be used. In either case, the syntax of the pattern appearing within the final pair of braces obeys the rules described below.

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

4.4.1.1 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:

\[
\begin{align*}
\text{species} \in \{\text{“platypus”}\} \\
\text{species} \in \{\text{“platypus”, “kangaroo”}\} \\
\text{species} \in \{\text{“platypus”, “kangaroo”, “wombat”}\}
\end{align*}
\]

The first example constrains the runtime value of the \text{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 trade-names (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 86.

4.4.1.2 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:

\[
\begin{align*}
\text{string_attr matches} \{/\text{regular expression}/\} \\
\text{string_attr matches} \{=~ /\text{regular expression}/\} \\
\text{string_attr matches} \{!~ /\text{regular expression}/\}
\end{align*}
\]

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:

\[
\begin{align*}
\text{units} \in \{/\text{km}/\text{h}|\text{mi}/\text{h}/\}\ \\
\text{units} \in \{^\text{km}/\text{h}|\text{mi}/\text{h}^\}
\end{align*}
\]

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

The regular expression patterns supported in cADL are as follows.

**Atomic Items**

- match any single character. E.g. / . . . / matches any 3 characters which occur with a space before and after;
[xyz] match any of the characters in the set xyz (case sensitive). E.g. /[0-9]/ matches any string containing a single decimal digit;

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

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

Grouping

(parentheses) 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”, “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.

### 4.4.2 Constraints on Integer

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

#### 4.4.2.1 List of Integers

Lists of integers expressed in the syntax from dADL (described in Lists of Built-in Types on page 34) 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.

4.4.2.2  Interval of Integer

Integer intervals are expressed using the interval syntax from dADL (described in Intervals of Ordered Primitive Types on page 33). Examples 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 {|<10|}  -- allow up to 9
length matches {|>10|}  -- allow 11 or more
length matches {|>=10|}  -- allow up to 10
length matches {|=10|}  -- allow 10 or more
length matches {|100+/-5|}  -- allow 100 +/- 5, i.e. 95 - 105
rate matches {|0..infinity|}  -- allow 0 - infinity, i.e. same as >= 0

dependant_range matches {|5.9; 101.110|}

4.4.3  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 \in \{5.5\}  -- fixed value
magnitude \in \{(5.5)\}  -- point interval (=fixed value)
magnitude \in \{(5.5..6.0)\}  -- interval
magnitude \in \{5.5, 6.0, 6.5\}  -- list
magnitude \in \{{<10.0}\}  -- allow anything less than 10.0
magnitude \in \{{>10.0}\}  -- allow greater than 10.0
magnitude \in \{{=10.0}\}  -- allow up to 10.0
magnitude \in \{{>=10.0}\}  -- allow 10.0 or more
magnitude \in \{{80.0+/-12.0\}}  -- allow 80 +/- 12

4.4.4  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}

4.4.5  Constraints on Character

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

4.4.5.1  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'}

4.4.5.2  Regular Expression

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

### 4.4.6 Constraints on Dates, Times and Durations

To Be Determined: sub-second precision not yet described

Dates, times, date/times and durations may all be constrained in three ways: using a list of strings, 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.

#### 4.4.6.1 Patterns

**Date, Time and Date/Time**

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. The following table shows the valid patterns which can be used, and the types implied by each pattern. These are formed from the abstract pattern `yyyy-mm-dd hh:mm:ss` (itself formed by translating each field of an ISO 8601 date/time into a letter representing its type), with either ‘?’ (meaning optional) or ‘X’ (not allowed) characters substituted in appropriate places.

<table>
<thead>
<tr>
<th>Implied Type</th>
<th>Pattern</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>yyyy-mm-dd</td>
<td>full date must be specified</td>
</tr>
<tr>
<td>Date, Partial Date</td>
<td>yyyy-mm-??</td>
<td>optional day;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. day in month forgotten</td>
</tr>
<tr>
<td>Date, Partial Date</td>
<td>yyyy-??-??</td>
<td>optional month, day;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i.e. any date allowed;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. mental health questionnaires which include</td>
</tr>
<tr>
<td></td>
<td></td>
<td>well known historical dates</td>
</tr>
<tr>
<td>Partial Date</td>
<td>yyyy-??-XX</td>
<td>optional month, no day;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(any examples?)</td>
</tr>
<tr>
<td>Time</td>
<td>hh:mm:ss</td>
<td>full time must be specified</td>
</tr>
<tr>
<td>Partial Time</td>
<td>hh:mm:XX</td>
<td>no seconds;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. appointment time</td>
</tr>
<tr>
<td>Partial Time</td>
<td>hh:??:XX</td>
<td>optional minutes, no seconds;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. normal clock times</td>
</tr>
<tr>
<td>Time, Partial Time</td>
<td>hh:???:??</td>
<td>optional minutes, seconds;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i.e. any time allowed</td>
</tr>
<tr>
<td>Date/Time</td>
<td>yyyy-mm-dd hh:mm:ss</td>
<td>full date/time must be specified</td>
</tr>
<tr>
<td>Date/Time, Partial Date/Time</td>
<td>yyyy-mm-dd hh:mm:??</td>
<td>optional seconds;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. appointment date/time</td>
</tr>
</tbody>
</table>
cADL - Constraint ADL
Rev 1.3.1

Duration Constraints

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

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

4.4.6.2 List and Intervals of Dates, Times, etc

Dates, times, date/times and durations can also be constrained using lists and intervals. Each date, time etc in such a list or 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
- |2004-05-20 00:00:00..2005-05-19 23:59:59| -- a date/time range

4.4.7 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 for complex types), 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.

4.4.8 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 contain an optional protocol section 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...
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 optionally 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; 1800-01-01 00:00:00}

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.

### 4.5 Syntax Specification

The cADL grammar is available as an [HTML document](http://my.openehr.org/wsvn/ref_impl_eiffel/TRUNK/components/adl_parser/src/syntax/cadl/parser/?rev=0&sc=0). 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://svn.openehr.org/ref_impl_eiffel](http://svn.openehr.org/ref_impl_eiffel). The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

#### 4.5.1 Grammar

The following provides the cADL parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository ([http://svn.openehr.org/ref_impl_eiffel](http://svn.openehr.org/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.

```plaintext
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:
    V_TYPE_IDENTIFIER
    | V_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_coded_term
  | c_ordinal
  | c_primitive_object
  | V_C_DOMAIN_TYPE
  | ERR_C_DOMAIN_TYPE
  | error

**archetype_internal_ref:**
  - SYM_USE_NODE V_TYPE_IDENTIFIER object_path
  | SYM_USE_NODE V_TYPE_IDENTIFIER error

**archetype_slot:**
  - c_archetype_slot_head SYM_MATCHES SYM_START_CBLOCK c_includes c_excludes SYM_END_CBLOCK

**c_archetype_slot_head:**
  - c_archetype_slot_id c_occurrences

**c_archetype_slot_id:**
  - SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER
  | SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER V_LOCAL_TERM_CODE_REF
  | SYM_ALLOW_ARCHETYPE error

**c_primitive_object:**
  - c_primitive

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

**c_any:**
  - *

**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_spec:
| V_ISO8601_DATE_CONSTRAINT_PATTERN
| date_value
| date_interval_value

c_date:
| c_date_spec
| c_date_spec ; date_value
| c_date_spec ; error

c_time_spec:
| V_ISO8601_TIME_CONSTRAINT_PATTERN
| time_value
| time_interval_value

c_time:
| c_time_spec
| c_time_spec ; time_value
| c_time_spec ; error

c_date_time_spec:
| V_ISO8601_DATE_TIME_CONSTRAINT_PATTERN
| date_time_value
| date_time_interval_value

c_date_time:
| c_date_time_spec
| c_date_time_spec ; date_time_value
| c_date_time_spec ; error

c_duration_spec:
| duration_value
| duration_interval_value

c_duration:
| c_duration_spec
| c_duration_spec ; duration_value
| c_duration_spec ; error

c_string_spec:
| V_STRING
| string_list_value
| string_list_value , SYM_LIST_CONTINUE
4.5.2 Symbols
The following shows the lexical specification for the cADL grammar.

```
---------/* comments */-----------------------------------------------
"--".*   -- Ignore comments
"--".*\n[ \t\r]*
---------/* symbols */-----------------------------------------------
"-"      -- -> Minus_code
"+"      -- -> Plus_code
```
cADL - Constraint ADL

Rev 1.3.1

"*" -- → 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 */ --------------------------------------
[MaTcHeSS] -- → SYM_MATCHES
[IiISIN] -- → 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][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][Ee][Ss][Tt][Ss] -- → SYM_EXISTS
---/* cADL keywords */---------------------------------------
[\[Ee\][Xx][Ii][Ss][Tt][Ee][Nn][Cc][Ee]] -- -> SYM_EXISTENCE
[\[Oo\][Cc][Cc][Uu][Rr][Ee][Nn][Cc][Ee][Ss]] -- -> SYM_OCCURRENCES
[\[Cc\][Aa][Rr][Dd][Ii][Nn][Aa][Ll][Yy]] -- -> SYM_CARDINALITY
[\[Oo\][Rr][Ee][Dd]] -- -> SYM_ORDERED
[\[Uu\][Nn][Oo][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][Oo][Ww][Aa][Rr][Cc][Hh][Ee][Tt][Yy][Pp][Ee]] -- -> SYM_ALLOW_ARCHETYPE
[\[Ii\][Nn][Cc][Ll][Uu][Dd][Ee]] -- -> SYM_INCLUDE
[\[Ee\][Cc][Ll][Uu][Dd][Ee]] -- -> SYM_EXCLUDE
---/* V_URI */-----------------------------------------------
[a-z]+://[^> 
]*
---/* V_QUALIFIED_TERM_CODE_REF */-----------------------------
-- any qualified code, e.g. [local::at0001], [local::ac0001], [loinc::700-0]
--
\[[a-zA-Z0-9().\-\]+::\[[a-zA-Z0-9._\-\]+\]
---/* 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]*
<IN_TERM_CONSTRAINT>\[ \t\]*[a-zA-Z0-9.\-\]+[ \t\]*[ \t\n]* -- pick up , line
<IN_TERM_CONSTRAINT>\[ \t\]*[a-zA-Z0-9.\-\]+[ \t\]*,\[ \t\n]* -- pick up ; line
<IN_TERM_CONSTRAINT>\[ \t\]*[a-zA-Z0-9.\-\]+\[ \t\n]* -- do nothing
<IN_TERM_CONSTRAINT>\[ \t\]*[a-zA-Z0-9.\-\]+\[ \t\n]*\] -- pick up ] line
--- V_LOCAL_TERM_CODE_REF */ ----------------------------------
-- any unqualified code, e.g. [at0001], [ac0001], [700-0]
\[[a-zA-Z0-9][a-zA-Z0-9._-]*\]

--- V_LOCAL_CODE */ ----------------------------------
\[a-zA-Z0-9.]+\]

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

--- V_ISO8601_DURATION */ ----------------------------------
P\([0-9]+[dD]?\([0-9]+[hH]?\([0-9]+[mM]?\([0-9]+[sS]??\)?\)?\]

--- 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?\]-\[dD?\][dD?\]:\[hH?X\][hH?X\]:\[mM?X\][mM?X\]:\[sS?X\][sS?X\]

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

--- V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
\[a-zA-Z\][a-zA-Z0-9-]*

--- 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
-- containing "()" 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

\[A-Z\][a-zA-Z0-9-]*[ \n]*<
-- match a pattern like
-- 'Type-Identifier whitespace '<

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

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

<IN_C_DOMAIN_TYPE>[\^>]*([ \n])*n
-- match up to next ')') not
-- preceded by a '>'

--- V_REGEXP */
The Archetype Definition Language (ADL) cADL - Constraint ADL
Rev 1.3.1

{mini-parser specification}
"{/" -- start of regexp
\^[^/\n]*\/{ -- match any segments with quoted slashes
\^[^/\n]*\}/ -- match final segment
\^[^/\n]*\} -- regexp formed using '^' delimiters

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

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

---------/* V_STRING */-----------------------------------------------
\"[^\\n"]*\" -- beginning of a string
\<IN_STR\\\n\[ ) -- \ - append '\'
\<IN_STR\\\n\[ ] -- \" - append '"'
\<IN_STR&[a-zA-Z][a-zA-Z0-9_]*; -- match ISO special character
\<IN_STR&#x([a-fA-F0-9_])4; -- match W3C XML special character
\<IN_STR[^\n"]*\n\[ ) -- match LF in line
\<IN_STR[^\n"]*\n\[ ] -- match final end of string
\<IN_STR\.|\n\] -- Error
The Archetype Definition Language (ADL) Assertions

5 Assertions

5.1 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 invariant section. The following simple assertion in an invariant clause says that the speed in kilometres of some node is related to the speed-in-miles by a factor of 1.6:

```
invariant
validity: /[at0001]/speed[at0002]/kilometres/magnitude = //[at0003]/speed[at0004]/miles/magnitude * 1.6
```

The archetype assertion language is a small language of its own. Formally it is a first-order predicate logic with equality and comparison operators (\(=\), \(>\), etc). It is very nearly a subset of the OMG’s emerging OCL (Object Constraint Language) syntax, and is very similar to the assertion syntax which has been used in the Object-Z [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.)

5.2 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.

<table>
<thead>
<tr>
<th>Textual Rendering</th>
<th>Symbolic Rendering</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches, is_in</td>
<td>(\in)</td>
<td>Set membership, “(p) is in (P)”</td>
</tr>
<tr>
<td>exists</td>
<td>(\exists)</td>
<td>Existential quantifier, “there exists ...”</td>
</tr>
<tr>
<td>for_all</td>
<td>(\forall)</td>
<td>Universal quantifier, “for all (x)...”</td>
</tr>
<tr>
<td>implies</td>
<td>(\Rightarrow), (\rightarrow)</td>
<td>Material implication, “(p) implies (q)”, or “if (p) then (q)”</td>
</tr>
<tr>
<td>and</td>
<td>(\land)</td>
<td>Logical conjunction, “(p) and (q)”</td>
</tr>
<tr>
<td>or</td>
<td>(\lor)</td>
<td>Logical disjunction, “(p) or (q)”</td>
</tr>
<tr>
<td>xor</td>
<td>(\lor)</td>
<td>Exclusive or, “only one of (p) or (q)”</td>
</tr>
<tr>
<td>not, ~</td>
<td>(\lnot), (\neg)</td>
<td>Negation, “not (p)”</td>
</tr>
</tbody>
</table>

The not operator can be applied as a prefix operator to all other operators except `for_all`; either textual rendering “not” or “\(\neg\)” can be used.

5.3 Operators

Assertion expressions can include arithmetic, relational and boolean operators, plus the existential and universal quantifiers.
5.3.1 Arithmetic Operators
The supported arithmetic operators are as follows:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Exponent: ^
- Modulo Division: % - remainder after integer division

5.3.2 Equality Operators
The supported equality operators are as follows:

- Equality: =
- Inequality: <>

The semantics of these operators are of value comparison.

5.3.3 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. Their domain is limited to values of comparable types.

5.3.4 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.

5.3.5 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.
5.4 Operands

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

- **manifest constant**: any constant of any primitive type, expressed according to the dADL syntax for values
- **variable reference**: any name starting with ‘$’, e.g. $body_weight;
- **object reference**: a path referring to an object node, i.e. any path ending in a node identifier
- **property reference**: a path referring to a property, i.e. any path ending in “.property_name”

If an assertion is used in an archetype slot definition, its paths refer to the archetype filling the slot, not the one containing the slot.

5.5 Precedence and Parentheses

To Be Continued:

5.6 Future

5.6.1 Variables

To Be Determined: main problem of variables is that they must have names, which are language-dependent; imagine if there were a mixture of variables added by authors in different languages. The only solution is to name them with terms.

To Be Determined: Variables have to be treated as term coordinations, and should be coded e.g. using ccNNNN codes (“cc” = coordinated code). Then they can be given meanings in any language.

**Predefined Variables**

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

- `$current_date`: returns the date whenever the archetype is evaluated
- `$current_time`: returns time whenever the archetype is evaluated
- `$current_date_time`: returns date/time whenever the archetype is evaluated

To Be Continued: these should be coded as well, using openEHR codes

**Archetype-defined Variables**

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

```
let $var_name = reference
```

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

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

```
invariant
  let $sys_bp =
    /[at0000]/data[at9001]/events[at9002]/data[at1000]/items[at1100]
  let $dia_bp =
    /[at0000]/data[at9001]/events[at9002]/data[at1000]/items[at1200]
```
$sys_{bp} \geq$ $dia_{bp}$

To Be Continued:
6 ADL Paths

6.1 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:

```
[/]attr_name ([object_id] [/])*  
```

Essentially ADL paths consist of segments separated by slashes (’/’), where each segment is an attribute name with optional object identifier, indicated by brackets (’[]’). A path either finishes in a slash, and identifies an object node, or finishes in an attribute name, and identifies an attribute node.

**ADL Paths** are formed from an alternation of segments made up of an attribute name and optional object node identifier, 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.**

6.2 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[@node_id = ‘at0001’] -- the member of ‘items’ with key ‘at0001’
```
In the above, `meaning()` is a notional function is defined for XPath in openEHR, 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.

### 6.3 Path Syntax

The ADL path grammar 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://my.openehr.org/wsvn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path/?rev=0&sc=0](http://my.openehr.org/wsvn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path/?rev=0&sc=0). The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

#### 6.3.1 Grammar

The following provides the ADL path parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository ([http://svn.openehr.org/ref_impl_eiffel](http://svn.openehr.org/ref_impl_eiffel)).

```
input:
  movable_path
  | location_path
  | error

location_path:
  absolute_path
  | absolute_path / call_path_segment
  | relative_path
  | relative_path / call_path_segment

movable_path:
  SYM_MOVABLE_LEADER 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

call_path_segment:
  V_FEATURE_CALL_IDENTIFIER
```

#### 6.3.2 Symbols

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

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

© 2003, 2004 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
“/”  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_TYPE_IDENTIFIER
[a-z][a-zA-Z0-9-]*[ ]\(\)  V_FEATURE_CALL_IDENTIFIER
[a-z][a-zA-Z0-9-]*  V_ATTRIBUTE_IDENTIFIER
7 ADL - Archetype Definition Language

This section describes ADL archetypes as a whole, adding a small amount of detail to the descriptions of dADL and cADL already given. The important topic of the relationship of the cADL-encoded definition section and the dADL-encoded ontology section is discussed in detail. In this section, only standard ADL (i.e. the cADL and dADL constructs and types described so far) is assumed. Archetypes for use in particular domains can also be built with more efficient syntax and domain-specific types, as described in Customising ADL on page 95, 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
invariant
    assertions
ontology
    dADL definitions section
    [revision_history
        dADL section]
```

7.1 Basics

7.1.1 Keywords

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

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

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

7.1.2 Node Identification

In the definition section of an ADL archetype, a particular scheme of codes is used for node identifiers as well as for denoting constraints on textual (i.e. language dependent) items. Codes are either local to the archetype, or from an external lexicon. This means that the archetype description is the same in all languages, and is available in any language that the codes have been translated to. All term codes are shown in brackets ([]). Codes used as node identifiers and defined within the same archetype are prefixed with “at” and 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.
7.1.3 Local Constraint Codes
A second kind of local code is used to stand for constraints on textual items in the body of the archetype. Although these could be included in the main archetype body, because they are language- and/or terminology-sensitive, they are defined in the ontology section, and referenced by codes prefixed by “ac”, e.g. [ac0009]. 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 7.5.4

7.2 Header Sections

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

```
archetype (adl_version=1.2)
mayo.openehr-ehr-entry.haematology.v1
```

The multi-axial identifier identifies archetypes in a global space. The syntax of the identifier is described under Archetype Identification on page 15 in The openEHR Archetype System.

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

```
archetype (adl_version=1.2; controlled)
mayo.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.

7.2.3 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=1.2)
mayo.openehr-ehr-entry.haematology-cbc.v1
specialise
mayo.openehr-ehr-entry.haematology.v1
```

Here the identifier of the new archetype is derived from that of the parent by adding a new section to its domain concept section. See Archetype Identification on page 15 in The openEHR Archetype System.

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

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

[at0010] -- haematology result

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

7.2.5 Language Section and Language Translation

The language section includes data describing the original language in which the archetype was authored (essential for evaluating natural language quality), and the total list of languages available in the archetype. There can be only one original_language. The translations list must be updated every time a translation of the archetype is undertaken. The following shows a typical example.

```plaintext
language
  original_language = <"en">
  translations = <
    ["de"] = <
      provenance = <"freddy@something.somewhere.co.uk">
      quality_control = <"British Medical Translator id 00400595">
    >
    ["ru"] = <
      provenance = <"vladimir@something.somewhere.ru">
      quality_control = <"Russion Translator id 892230A">
    >

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.

7.2.6 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.

```plaintext
description
  original_author = <
    ["name"] = <"Dr J Joyce">
    ["organisation"] = <"NT Health Service">
    ["date"] = <2003-08-03>
  >
  lifecycle_state = <"initial">

details = <
  ["en"] = <
    purpose = <"archetype for diabetic patient review">
    use = <"used for all hospital or clinic-based diabetic reviews, including first time. Optional sections are removed according to the particular review">
    misuse = <"not appropriate for pre-diagnosis use">
    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 HL7 Templates Proposal [17] and the meta-data of the SynEx and GeHR archetypes.

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

To Be Determined: will this be specified here or in a “health informatics profile” for ADL? See Archetype Object Model doc.

7.3 Definition Section

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

```cADL
definition
ENTRY[at0000] ∈ { -- blood pressure measurement
    name ∈ {
        CODED_TEXT ∈ {
            code ∈ {
                CODE_PHRASE ∈ {[ac0001]}
            }
        }
    }

    data ∈ {
        HISTORY[at9001] ∈ { -- history
            events cardinality ∈ {1..*} ∈ {
                EVENT[at9002] occurrences ∈ {0..1} ∈ {-- baseline
                    name ∈ {
                        CODED_TEXT ∈ {
                            code ∈ {
                                CODE_PHRASE ∈ {[ac0002]}
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
-- systemic arterial BP
```
items cardinality ∈ {2..*} ∈ {
ELEMENT[at1100] ∈ {
    -- systolic BP
    name ∈ {
        -- any synonym of 'systolic'
        CODED_TEXT ∈ {
            code ∈ {
                CODE_PHRASE ∈ {[ac0002]}
            }
        }
    }
}
value ∈ {
    QUANTITY ∈ {
        magnitude ∈ {0..1000}
        property ∈ {[properties::0944]}
            -- “pressure”
        units ∈ {[units::387]} -- “mm[Hg]”
    }
}
ELEMENT[at1200] ∈ {
    -- diastolic BP
    name ∈ {
        -- any synonym of 'diastolic'
        CODED_TEXT ∈ {
            code ∈ {
                CODE_PHRASE ∈ {[ac0003]}
            }
        }
    }
}
value ∈ {
    QUANTITY ∈ {
        magnitude ∈ {0..1000}
        property ∈ {[properties::0944]}
            -- “pressure”
        units ∈ {[units::387]} -- “mm[Hg]”
    }
}
ELEMENT[at9000] occurrences ∈ {0..*} ∈ {*}
    -- unknown new item
}

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

### 7.4 Invariant Section

The invariant 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. Invariants are expressed in the archetype assertion language, described in section 5 on page 71.

An invariant statement is a first order predicate logic statement which can be evaluated to a boolean result at runtime. Objects and properties are referred to using paths.
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:

```plaintext
invariant

    validity: /[at0001]/speed[at0002]/kilometres/magnitude =
              /[at0003]/speed[at0004]/miles/magnitude * 1.6
```

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

### 7.5 Ontology Section

#### 7.5.1 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 general layout of this section.

```plaintext
ontology

    terminologies_available = <"snomed_ct", ...>

    term_definitions = <
        ["en"] = <
            _items = <...>
        >
        ["de"] = <
            _items = <...>
        >
    >

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

    constraint_definitions = <
        ["en"] = <...>
        ["de"] = <
            _items = <...>
        >
        ...
    >

    constraint_binding = <
        ["snomed_ct"] = <...>
        ...
    >
```

The `term_definitions` section is mandatory, and must be defined for each translation carried out. Each of these sections can have its own meta-data, which appears within description sub-sections, such as the one shown above providing translation details.
7.5.2 Ontology Header Statements
The terminologies_available statement includes the identifiers of all terminologies for which term_binding sections have been written.

7.5.3 Term_definition Section
This section is where all archetype local terms (that is, terms of the form `[atNNNN]`) are defined. The following example shows an extract from the English and German term definitions for the archetype local terms in a problem/SOAP headings archetype. Each term is defined using a structure of name/value pairs, and must at 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.

To Be Determined: the use of “items” in the following is historical in ADL, and will be changed in future to the proper form of dADL for nested containers, i.e. removing the “items = <” block altogether.

```xml
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:

```xml
["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 7.5.5 and section 7.5.6.

### 7.5.4 Constraint_definition Section

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

```xml
items = <
    ["at1015"] = <
        text = "type of hepatitis"
        description = "any term which means a kind of viral hepatitis"
    >
>
```

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

### 7.5.5 Term_binding Section

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

A typical term binding section resembles the following:

```xml
term_binding("umls") = <
    ["umls"] = <
        items =<
            ["at0000"] = [umls::0000000] -- 1-minute event
            ["at0002"] = [umls::0000001] -- 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
        >
    >
```
The Archetype Definition Language (ADL) - Archetype Definition Language
Rev 1.3.1

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

To Be Determined: future possibility: more than one binding to the same terminology for different purposes, or by different authors?
To Be Determined: need to handle numerous small vocabularies defined by one authority e.g. HL7.

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

```
constraint_binding = <
    ["snomed_ct"]
    _items = <
        ["ac0001"] = <http://terminology.org?terminology_id=snomed_ct&&
            has_relation=[102002];with_target=[128004]>
        ["ac0002"] = <http://terminology.org?terminology_id=snomed_ct&&
            synonym_of=[128025]>
   >
>
```

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

7.6 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"
        >
```
7.7 Validity Rules

This section describes the formal (i.e. checkable) semantics of ADL archetypes. It is recommended that parsing tools use the identifiers published here in their error messages, as an aid to archetype designers.

7.7.1 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 openEHR 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 cADL syntax string, or in an equivalent plug-in syntax.

**VARDT: archetype definition typename validity.** The topmost typename mentioned in the archetype definition section must match the type mentioned in the type-name slot of the first segment of the archetype id.

7.7.2 Coded Term Validity

All node identifiers (‘at’ codes) used in the *definition* part of the archetype must be defined in the *term_definitions* part of the ontology.

**VATDF: archetype term validity.** Each archetype term used as a node identifier the archetype definition must be defined in the *term_definitions* part of the ontology.
All constraint identifiers (‘ac’ codes) used in the definition part of the archetype must be defined in the constraint_definitions part of the ontology.

**VACDF: node identifier validity.** Each constraint code used in the archetype definition part must be defined in the constraint_definitions part of the ontology.

### 7.7.3 Definition Section

The following constraints apply to the definition section of the archetype.

**VDFAI: archetype identifier validity in definition.** Any archetype identifier mentioned in an archetype slot in the definition section must conform to the published openEHR specification for archetype identifiers.

**VDFPT: path validity in definition.** Any path mentioned in the definition section must be valid syntactically, and a valid path with respect to the hierarchical structure of the definition section.

### 7.8 Syntax Specification

#### 7.8.1 Grammar

This section describes the ADL grammar, as implemented and tested in the openEHR implementation project.

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

archetype: arch_identification
  arch_specialisation
  arch_concept
  arch_description
  arch_definition
  arch_invariant
  arch_ontology

arch_identification: SYM_ARCHETYPE V_ARCHETYPE_ID | SYM_ARCHETYPE error

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

arch_concept: SYM_CONCEPT V_LOCAL_TERM_CODE_REF | SYM_CONCEPT error

arch_description: -- no meta-data ok | SYM_DESCRIPTION V_DADL_TEXT | SYM_DESCRIPTION error

```
7.8.2 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][Ss][Cc][Rr][Ii][Pp][Tt][Ii][Oo][Nn][ \t\r]*\n SYM_DEFINITION
 -- mini-parser to generate V_DADL_TEXT
^[Dd][Ee][Su][Cc][Rr][Ii][Pp][Tt][Ii][Oo][Nn][ \t\r]*\n SYM_DESCRIPTION
 -- mini-parser to generate V_CADL_TEXT
^[Ii][Nn][Vv][Aa][Rr][Aa][Nn][Tt][ \t\r]*\n SYM_INVARIANT
 -- mini-parser to generate V_ASSERTION_TEXT
^[Oo][Nn][Tt][Oo][Ll][Oo][Gg][Yy][ \t\r]*\n SYM_ONTOLOGY
 -- mini-parser to generate V_DADL_TEXT
```

```
---/* term code reference */ ----------------------------------------
\([a-zA-Z0-9][a-zA-Z0-9.-]*\) V_LOCAL_TERM_CODE_REF
```
The Archetype Definition Language (ADL)

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

---/* identifiers */----------------------------------------------
[a-zA-Z][a-zA-Z0-9-_]*
V_IDENTIFIER
8 The ADL Parsing Process

8.1 Overview

FIGURE 6 illustrates the ADL parsing process. An ADL file is converted by the ADL parser into an ADL parse tree. This tree is an in-memory object structure representation of the semantics of the archetype, in a form corresponding to the openEHR Archetype Object Model. This model is then validated by the semantic checker of the ADL parser, which can verify numerous things, such as that term codes referenced in the definition section are defined in the ontology section. It can also validate the classes and attributes mentioned in the archetype against a specification for the relevant information model (e.g. in XMI or some equivalent).

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

- **C_complex_object**: any interior node representing a constraint on instances of some non-primitive type, e.g. ENTRY, SECTION;
- **C_attribute**: a node representing a constraint on an attribute (i.e. UML ‘relationship’ or ‘primitive attribute’) in an object type;

**FIGURE 6** Parsed ADL Structure
C_primitive_object: a node representing a constraint on a primitive (built-in) object type;

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

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

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

See the openEHR Archetype Object Model (AOM) for details.
9 Customising ADL

9.1 Introduction

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

9.1.1 Custom Syntax

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

```
standard ADL using type and attribute names

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

clinical ADL syntax

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

FIGURE 7 Constraints using additional syntax

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

9.1.2 Custom Constraint Classes

Another situation in which standard ADL falls short is when the required semantics of constraint are different from those provided by the standard approach. Consider a simple type QUANTITY, shown at the top of FIGURE 8, which could be used to represent a person’s age in data. A typical ADL constraint to enable QUANTITY to be used to represent age in clinical data is shown below, followed by its
expression in ADL. The only way to do this in ADL is to use multiple alternatives. While this is a perfectly legal approach, it makes processing by software difficult, since the way such a constraint would be displayed in a GUI would be factored differently.

<table>
<thead>
<tr>
<th>reference model type</th>
<th>QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>property: String</td>
<td></td>
</tr>
<tr>
<td>magnitude: Real</td>
<td></td>
</tr>
<tr>
<td>units: String</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>desired constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>property matches “time”</td>
</tr>
<tr>
<td>units matches “years” or “months”</td>
</tr>
<tr>
<td>if units is “years” then magnitude matches 0..200</td>
</tr>
<tr>
<td>if units is “months” then magnitude matches 3..36</td>
</tr>
<tr>
<td>etc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>standard ADL expression using alternates</th>
</tr>
</thead>
<tbody>
<tr>
<td>age matches {</td>
</tr>
<tr>
<td>QUANTITY matches {</td>
</tr>
<tr>
<td>property matches {“time”}</td>
</tr>
<tr>
<td>units matches {“years”}</td>
</tr>
<tr>
<td>magnitude matches {0.0..200.0}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>QUANTITY matches {</td>
</tr>
<tr>
<td>property matches {“time”}</td>
</tr>
<tr>
<td>units matches {“months”}</td>
</tr>
<tr>
<td>magnitude matches {3.0..12.0}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

**FIGURE 8** Standard ADL for Constraint on a Quantity Class

A more powerful possibility is to introduce a new class into the archetype model, representing the concept “constraint on QUANTITY”, which we will call C_QUANTITY here. Such a class fits into the class model of archetypes (described in the openEHR Archetype Model document), inheriting from the class C_DOMAIN_TYPE. The C_QUANTITY class is illustrated in FIGURE 9, and corresponds to the way constraints on QUANTITY objects are expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

<table>
<thead>
<tr>
<th>custom constraint type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_QUANTITY</td>
</tr>
<tr>
<td>property: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_QUANTITY_ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>units[1]: String</td>
</tr>
<tr>
<td>magnitude[0..1]: Interval&lt;Real&gt;</td>
</tr>
</tbody>
</table>

**FIGURE 9** Custom Constraint Type for QUANTITY

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

```
value matches { 
  C\_QUANTITY < 
    property = "time"
    list = <
      items = <
        [1] = <
          units = "yr"
          magnitude = <|0.0..200.0|>
        >
        [2] = <
          units = "mth"
          magnitude = <|1.0..36.0|>
        >
      >
    >
  }
}
```

\textbf{FIGURE 10} Inclusion of a Constraint Object as Data

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

- the dADL section occurs inside the \{\} block where its standard ADL equivalent would have occurred (i.e. no other delimiters or special marks are needed);
- the dADL section must be ‘typed’, i.e. it must start with a type name, which should be a rule-based transform of a reference model type (as described in Adding Type Information on page 28);
- the dADL instance must obey the semantics of the custom type of which it is an instance.

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

10.1 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.

10.2 Constraint Syntaxes

10.2.1 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.

10.3 Ontology Formalisms

10.3.1 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
Relationship of ADL to Other Formalisms

The Archetype Definition Language (ADL)
Rev 1.3.1

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.
Lastly, OWL (as of mid 2004) is still under development, and has only a very limited set of primitive constraint types (it is not possible for example to state a constraint on an Integer attribute of the form ‘any value between 80 and 110’), although this is being addressed; by contrast, ADL provides a rich set of constraints on primitive types, including dates and times.

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).

10.3.2 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
```
Relationship of ADL to Other Formalisms

The Archetype Definition Language (ADL)

Rev 1.3.1

(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].

10.4 XML-based Formalisms

10.4.1 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 openEHR 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.
A References

Publications


Resources

20. SynEx project, UCL. http://www.chime.ucl.ac.uk/HealthI/SynEx/.
END OF DOCUMENT
The Archetype Definition Language Version 2 (ADL2)

Editors: {T Beale, S Heard}

Revision: 2.0

Pages: 109

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

© 2003-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman

David Ingram, Professor of Health Informatics, CHIME, University College London

Founding Members

Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
## Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RELEASE 1.0</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.96</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE 0.95</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>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. <strong>Remove UML models to “Archetype Object Model” document.</strong> 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.</td>
<td>A Rector</td>
<td>15 Nov 2004</td>
</tr>
<tr>
<td><strong>RELEASE 0.9</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>CR-000079. Change interval syntax in ADL.</td>
<td>T Beale</td>
<td>24 Jan 2004</td>
</tr>
<tr>
<td>Issue</td>
<td>Details</td>
<td>Raiser</td>
<td>Completed</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>0.9.9</td>
<td>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.</td>
<td>T Beale, S Heard</td>
<td>28 Dec 2003</td>
</tr>
<tr>
<td>0.9.8</td>
<td>CR-000070. Create Archetype System Description. Moved Archetype Identification Section to new Archetype System document. Copyright Assigned by Ocean Informatics P/L Australia to The openEHR Foundation.</td>
<td>T Beale, S Heard</td>
<td>29 Nov 2003</td>
</tr>
<tr>
<td>0.9.7</td>
<td>Added simple value list continuation (&quot;...&quot;). Changed path syntax so that trailing ‘/’ required for object paths. Remove ranges with excluded limits. Added terms and term lists to dADL leaf types.</td>
<td>T Beale</td>
<td>01 Nov 2003</td>
</tr>
<tr>
<td>0.9.6</td>
<td>Additions during HL7 WGM Memphis Sept 2003</td>
<td>T Beale</td>
<td>09 Sep 2003</td>
</tr>
<tr>
<td>0.9.5</td>
<td>Added comparison to other formalisms. Renamed CDL to cADL and dDL to dADL. Changed path syntax to conform (nearly) to XPath. Numerous small changes.</td>
<td>T Beale</td>
<td>03 Sep 2003</td>
</tr>
<tr>
<td>0.9</td>
<td>Rewritten with sections on cADL and dDL.</td>
<td>T Beale</td>
<td>28 July 2003</td>
</tr>
<tr>
<td>0.8.1</td>
<td>Added basic type constraints, re-arranged sections.</td>
<td>T Beale</td>
<td>15 July 2003</td>
</tr>
<tr>
<td>0.8</td>
<td>Initial Writing</td>
<td>T Beale</td>
<td>10 July 2003</td>
</tr>
</tbody>
</table>
Trademarks

“Microsoft” and “.Net” are registered trademarks of the Microsoft Corporation.

“Java” is a registered trademark of Sun Microsystems.

“Linux” is a registered trademark of Linus Torvalds.

Acknowledgements

Sebastian Garde, Central Qld University, Australia, for german translations.
# Table of Contents

## 1 Introduction
- Purpose .............................................................................................................. 11
- Related Documents .......................................................................................... 11
- Nomenclature .................................................................................................... 11
- Status .................................................................................................................. 11
- Peer review ........................................................................................................ 11
- Conformance ...................................................................................................... 12

## 2 Overview
- What is ADL? ........................................................................................................ 13
  - Structure ............................................................................................................ 13
  - An Example ...................................................................................................... 14
  - Semantics .......................................................................................................... 15
- Computational Context ....................................................................................... 15
- XML form of Archetypes .................................................................................... 16
- Changes From Previous Versions
  - Version 2.0 from Version 1.3 ......................................................................... 16
  - Version 1.3 from Version 1.2 ......................................................................... 17
  - Version 1.2 from Version 1.1 ......................................................................... 18
- Tools .................................................................................................................... 19

## 3 dADL - Data ADL
- Overview ............................................................................................................. 21
- Basics .................................................................................................................. 22
  - Scope of a dADL Document .......................................................................... 22
  - Keywords .......................................................................................................... 22
  - Reserved Characters ....................................................................................... 22
  - Comments ........................................................................................................ 23
  - Reserved Character Encoding ......................................................................... 23
  - Information Model Identifiers ....................................................................... 23
  - Semi-colons ...................................................................................................... 24
- Paths ..................................................................................................................... 24
- Structure .............................................................................................................. 24
  - General Form ................................................................................................... 24
    - Outer Delimiters ............................................................................................ 25
    - Paths .............................................................................................................. 25
  - Empty Sections ................................................................................................. 25
  - Container Objects ........................................................................................... 26
    - Paths .............................................................................................................. 28
  - Nested Container Objects ............................................................................... 28
    - Paths .............................................................................................................. 28
  - Adding Type Information ............................................................................... 28
  - Associations and Shared Objects .................................................................. 29
    - Paths .............................................................................................................. 30
- Leaf Data - Built-in Types .................................................................................. 30
  - Primitive Types ............................................................................................... 31
    - Character Data ............................................................................................. 31
    - String Data .................................................................................................. 31
    - Integer Data ................................................................................................. 31
4 cADL - Constraint ADL .............................................................. 47

4.1 Overview ..................................................................................... 47
4.2 Basics........................................................................................... 48
4.2.1 Keywords .................................................................................. 48
4.2.2 Comments .................................................................................. 49
4.2.3 Information Model Identifiers ....................................................... 49
4.2.4 Node Identifiers ......................................................................... 49
4.2.5 Natural Language ....................................................................... 49
4.3 Structure ...................................................................................... 49
4.3.1 Complex Objects ...................................................................... 50
4.3.2 Attribute Constraints ................................................................. 51
4.3.2.1 Existence .............................................................................. 51
4.3.3 Single-valued Attributes ............................................................. 52
4.3.4 Container Attributes ................................................................. 53
4.3.4.1 Cardinality .......................................................................... 53
4.3.4.2 Occurrences ....................................................................... 53
4.3.5 “Any” Constraints ................................................................... 54
4.3.6 Object Node Identification and Paths ........................................... 55
4.3.7 Internal References .................................................................. 57
4.3.8 Archetype Slots ....................................................................... 57
4.3.9 Mixed Structures ...................................................................... 59
4.4 Constraints on Primitive Types .................................................... 59
4.4.1 Constraints on String ................................................................ 60
4.4.1.1 List of Strings ..................................................................... 60
4.4.2 Constraints on Integer ............................................................... 61
4.4.2.1 List of Integers .................................................................... 61
4.4.3 Constraints on Real .................................................................. 62
4.4.4 Constraints on Boolean .............................................................. 62
4.4.5 Constraints on Character ............................................................ 62
4.4.5.1 List of Characters ................................................................. 62
4.4.6 Constraints on Dates, Times and Durations ............................... 63
4.4.6.1 Patterns .............................................................................. 63
4.4.6.2 List and Intervals of Dates, Times, etc ................................. 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.7</td>
<td>Constraints on Lists of Primitive types</td>
<td>64</td>
</tr>
<tr>
<td>4.4.8</td>
<td>Assumed Values</td>
<td>64</td>
</tr>
<tr>
<td>4.5</td>
<td>Syntax Specification</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Grammar</td>
<td>65</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Symbols</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>Assertions</td>
<td>75</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Keywords</td>
<td>75</td>
</tr>
<tr>
<td>5.3</td>
<td>Operators</td>
<td>75</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Arithmetic Operators</td>
<td>76</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Equality Operators</td>
<td>76</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Relational Operators</td>
<td>76</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Boolean Operators</td>
<td>76</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Quantifiers</td>
<td>76</td>
</tr>
<tr>
<td>5.4</td>
<td>Operands</td>
<td>77</td>
</tr>
<tr>
<td>5.5</td>
<td>Precedence and Parentheses</td>
<td>77</td>
</tr>
<tr>
<td>5.6</td>
<td>Syntax Specification</td>
<td>77</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Grammar</td>
<td>77</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Symbols</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>Declarations</td>
<td>79</td>
</tr>
<tr>
<td>6.1</td>
<td>Predefined Variables</td>
<td>79</td>
</tr>
<tr>
<td>6.2</td>
<td>Archetype-defined Variables</td>
<td>79</td>
</tr>
<tr>
<td>7</td>
<td>ADL Paths</td>
<td>81</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview</td>
<td>81</td>
</tr>
<tr>
<td>7.2</td>
<td>Relationship with W3C Xpath</td>
<td>81</td>
</tr>
<tr>
<td>7.3</td>
<td>Path Syntax</td>
<td>82</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Grammar</td>
<td>82</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Symbols</td>
<td>82</td>
</tr>
<tr>
<td>8</td>
<td>ADL - Archetype Definition Language</td>
<td>85</td>
</tr>
<tr>
<td>8.1</td>
<td>Basics</td>
<td>85</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Order of Sections</td>
<td>85</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Keywords</td>
<td>85</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Node Identification</td>
<td>86</td>
</tr>
<tr>
<td>8.1.4</td>
<td>Local Constraint Codes</td>
<td>86</td>
</tr>
<tr>
<td>8.2</td>
<td>Header Sections</td>
<td>86</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Archetype_id Section</td>
<td>86</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Adl_version Section</td>
<td>86</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Is_controlled Section</td>
<td>86</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Parent_archetype_id Section</td>
<td>87</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Concept Section</td>
<td>87</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Original_language and Translations Sections</td>
<td>87</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Description Section</td>
<td>88</td>
</tr>
<tr>
<td>8.3</td>
<td>Definition Section</td>
<td>89</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Design-time and Run-time paths</td>
<td>90</td>
</tr>
<tr>
<td>8.4</td>
<td>Invariant Section</td>
<td>90</td>
</tr>
<tr>
<td>8.5</td>
<td>Ontology Section</td>
<td>91</td>
</tr>
</tbody>
</table>
8.5.1 Overview ........................................................................................................... 91
8.5.2 Terminologies_available Sub-section ............................................................. 92
8.5.3 Term_definitions Sub-section ......................................................................... 92
8.5.4 Constraint_definitions Sub-section ............................................................... 93
8.5.5 Term_binding Sub-section .............................................................................. 93
8.5.6 Constraint_binding Sub-section ....................................................................... 94
8.6 Revision_history Section ................................................................................ 94
8.7 Validity Rules ..................................................................................................... 95
8.7.1 Global Archetype Validity ............................................................................ 95
8.7.2 Coded Term Validity .................................................................................... 95
8.7.3 Definition Section ......................................................................................... 96
9 The ADL Parsing Process ................................................................................. 97
9.1 Overview ........................................................................................................... 97
10 Customising ADL ......................................................................................... 99
10.1 Introduction ................................................................................................... 99
10.1.1 Custom Syntax ......................................................................................... 99
10.1.2 Custom Constraint Classes ....................................................................... 99
11 Relationship of ADL to Other Formalisms ............................................. 103
11.1 Overview ...................................................................................................... 103
11.2 Constraint Syntaxes ..................................................................................... 103
11.2.1 OCL (Object Constraint Language) ......................................................... 103
11.3 Ontology Formalisms .................................................................................. 103
11.3.1 OWL (Web Ontology Language) ............................................................. 103
11.3.2 KIF (Knowledge Interchange Format) .................................................. 105
11.4 XML-based Formalisms ............................................................................. 106
11.4.1 XML-schema ......................................................................................... 106
1 Introduction

1.1 Purpose

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

The intended audience includes:

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

1.2 Related Documents

Related documents include:

- The openEHR Archetype Object Model (AOM)
- The openEHR Archetype Profile (oAP)

1.3 Nomenclature

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

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://svn.openeehr.org/specification/TAGS/Release-1.0/publishing/architecture/am/adl2.pdf.

The latest version of this document can be found at http://svn.openeehr.org/specification/TRUNK/publishing/architecture/am/adl2.pdf.

Blue text indicates sections under active development.

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 artifact to an openEHR Reference Model specification is determined by a formal test of that artifact against the relevant openEHR 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.
2 Overview

2.1 What is ADL?

Archetype Definition Language (ADL) is a formal language for expressing archetypes, which are constraint-based models of domain content. The archetype concept is described by Beale [1], [2]. The openEHR Archetype Object Model [3] describes an object model equivalent of the ADL syntax. The openEHR archetype framework is described in terms of Archetype Definitions and Principles [6] and an Archetype System [7]. Other semantic formalisms which were considered in the course of archetype, and some which remain relevant are described in detailed in section 11 on page 103.

To describe constraints on data which are instances of some information model (e.g. expressed in UML), ADL archetypes use three syntaxes: dADL (a data expression syntax), cADL (a constraint expression syntax), and a version of first-order predicate logic (FOPL). It is most useful when very generic information models are used for describing the data in a system, for example, where the logical concepts PATIENT, DOCTOR and HOSPITAL might all be represented using a small number of classes such as PARTY and ADDRESS. In such cases, archetypes are used to constrain the valid structures of instances of these generic classes to represent the desired domain concepts. In this way future-proof information systems can be built - relatively simple information models and database schemas can be defined, and archetypes supply the semantic modelling, completely outside the software. ADL can thus be used to write archetypes for any domain where formal object model(s) exist which describe data instances.

When archetypes are used at runtime in particular contexts, they are composed into larger constraint structures, with local or specialist constraints added, via the use of templates. The formalism of templates is dADL. Archetypes can be specialised by creating an archetypes that reference existing archetypes as parents; such archetypes can only make certain changes while remaining compatible with the parent.

Another major function of archetypes is to connect information structures to formal terminologies. Archetypes are language-neutral, and can be authored in and translated into any language.

Finally, archetypes are completely path-addressable in a manner similar to XML data, using path expressions that are directly convertible to Xpath expressions.

2.1.1 Structure

An ADL archetype is syntactically a dADL document, containing a definition section expression in the cADL syntax. The structure of an ADL archetype is shown in FIGURE 1.
2.1.2 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 example 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_id = <“adl-test-instrument.guitar.draft”>
adl_version = <“2.0”>
concept = <[concept_code]>
original_language = <“en”>
transl

definition = <

INSTRUMENT[at0000] matches {
  size matches {{60..120}}  -- size in cm
  date_of_manufacture matches {yyyy-mm-??}  -- year & month ok

FIGURE 1 ADL Archetype Structure

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.
2.1.3 Semantics

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

While ADL syntax remains the primary abstract formalism for expressing archetypes, the AOM defines the semantics of an archetype, in particular relationships which must hold true between the parts of an archetype for it to be valid as a whole.

2.2 Computational Context

An archetypes is a structured model of domain content, such as “blood pressure”. Archetypes sit between knowledge resources in a computing environment, such as terminologies and ontologies, and runtime data in production systems. Their primary purpose is to provide a reusable, interoperable way
Overview

The Archetype Definition Language Version 2 (ADL2)

Rev 2.0

of managing data creation, validation and querying, by ensuring that data conform to particular structures and semantic constraints. Every ADL archetype is written with respect to a particular information model, often known as a “reference model”, if it is a shared, public specification.

Archetypes are applied to data via the use of templates, which are defined at a local level. Templates generally correspond closely to screen forms, and may be re-usable at a local or regional level. Templates do not introduce any new semantics to archetypes, they simply specify the use of particular archetypes in particular hierarchical compositions, as well as default data values.

A third artifact that governs the functioning of archetypes and templates at runtime is a local palette, which specifies which natural language(s) and terminologies are in use in the locale. The use of a palette removes irrelevant languages and terminology bindings from archetypes, retaining only those relevant to actual use. FIGURE 3 illustrates the overall environment in which archetypes, templates, and a locale palette exist.

2.3 XML form of Archetypes

With ADL parsing tools it is possible to convert ADL to any number of forms, including various XML formats. XML instance can be generated from the object form of an archetype in memory. An XML-schema corresponding to the ADL Object Model is published on openEHR.org.

2.4 Changes From Previous Versions

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

2.4.1 Version 2.0 from Version 1.3

In this major revision of ADL, the ADL syntax is synchronised properly with the openEHR Archetype Object Model, although the semantics remain virtually unchanged. Accordingly, all top-level
keywords in an ADL archetype are now considered the direct equivalent of an attribute of the same name in the AOM. To enable this, the entire ADL archetype is now a dADL document, with plug-in syntax sections for expressing constraints and invariants. The specific changes in this revision are:

- replace ‘archetype’ keyword with a dADL section named ‘archetype_id’ which has a String value;
- replace the ADL 1.2 adl_version syntax with a new top-level dADL section ‘adl_version’ which has a String value;
- replace the ADL 1.2 controlled syntax with a new top-level dADL section ‘is_controlled’ which has a Boolean value;
- replace ‘specialise’ keyword with a dADL section named ‘parent_archetype_id’;
- replace ‘language’ section with two top-level dADL sections ‘original_language’ and ‘translations’.

These changes facilitate human understanding as well as automated processing, particularly serialisation/deserialisation into and out of object form.

A full dADL form of an archetype can 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. This makes conversion to and from various XML formats trivial.

### 2.4.2 Version 1.3 from Version 1.2

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:

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

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

```plaintext
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.
Overview
The Archetype Definition Language Version 2 (ADL2)
Rev 2.0

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.

2.4.3 Version 1.2 from Version 1.1

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

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

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

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

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

This has been changed to look like the following:

```
school_schedule = <
  locations = <
    [1] = <...
    [2] = <...
    [3] = ARTS_PAVILLION<br>
  >
  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 <...
```

Date of Issue: 18 Jan 2006
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 archetypes 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.

2.5 Tools
A validating ADL parser is freely available from http://my.openehr.org/wsvn/oe_distrib/. It has been wrapped for use in Java and Microsoft .Net, and standard C/C++ environments. See the website for the latest status.
3 dADL - Data ADL

3.1 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:

```dADL
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;
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.

3.2 Basics

3.2.1 Scope of a dADL Document

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

3.2.2 Keywords

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

3.2.3 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:

```
`````

```
```
```
```
```

```
3.2.4 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.

3.2.5 Reserved Character Encoding
Reserved characters need to be encoded in some contexts to avoid parsing ambiguity. Encoding is needed in strings, characters and URIs. The standard for encoding special characters, quotes, and other reserved characters is the Extensible Markup Language (XML) 1.0 (Third Edition) standard ([http://www.w3.org/TR/REC-xml](http://www.w3.org/TR/REC-xml)). Encoding of special characters according to this standard is as follows:

- `&#NNN;` - decimal code for a character from ISO/IEC 10646
- `&#xNNN;` - hexadecimal code for a character from ISO/IEC 10646
- `&aaa;` - mnemonic code any character from ISO/IEC 10646, e.g. `&lt;` means ‘<’

In dADL, special encoding is needed in the following situations:

- **in string values** (e.g. “xxxx”): the ‘”’ (double quote) character and possibly other special characters (defined in the XML standard) must be encoded;
- **in character values** (e.g. ‘x’): the ‘’ (single quote) character and possibly other special characters (defined in the XML standard) must be encoded;
- **in URIs**: the <> characters, whitespace characters, and various others must be encoded according to the Uniform Resource Identifier (URI): Generic Syntax, Internet proposed standard RFC 3986, January 2005 (see [http://www.ietf.org/rfc/rfc3986.txt](http://www.ietf.org/rfc/rfc3986.txt)).

In summary, special character encoding follows the rule:

Some reserved characters must be encoded within dADL String, Character and URI values, using the W3C XML standard or IETF RFC 3986 as appropriate.

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

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

Type names in this document are in all uppercase, e.g. PERSON, 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 types assumed from constructed types.

Attribute names are shown in all lowercase, e.g. home_address. In both cases, underscores are used to represent word breaks. This convention is used to maximise the readability of this document, and other conventions may be used, such as the common programmer’s mixed-case convention exemplified by Person and homeAddress, as long as they obey the rule above. The convention chosen for any particular dADL document should be based on the convention used in the underlying information model. Identifiers are shown in green in this document.
3.2.7 Semi-colons
Semi-colons can be used to separate dADL blocks, for example when it is preferable to include multiple attribute/value pairs on one line. Semi-colons make no semantic difference at all, and are included only as a matter of taste. The following examples are equivalent:

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

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

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

Semi-colons are completely optional in dADL.

3.3 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 81. A path either finishes in a slash, and identifies an object node, or finishes in an attribute name, and identifies an attribute node. 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.

3.4 Structure
3.4.1 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 31) 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 3.4.6 on page 29.

### 3.4.1.1 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.**

### 3.4.1.2 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
```

### 3.4.2 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.
3.4.3 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 34 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 44 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.

```plaintext
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):

```plaintext
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
```

---

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
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).

### 3.4.3.1 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”
```

### 3.4.4 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”>
  >
>
```

### 3.4.4.1 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]
```

### 3.4.5 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.

\[
\text{destinations} = < \\
\quad ["seville"] = (\text{TOURIST\_DESTINATION}) < \\
\quad \text{profile} = (\text{DESTINATION\_PROFILE}) <> \\
\quad \text{hotels} = < \\
\quad \\
\quad ["gran sevilla"] = (\text{HISTORIC\_HOTEL}) <> \\
\quad ["sofitel"] = (\text{LUXURY\_HOTEL}) <> \\
\quad ["hotel real"] = (\text{PENSION}) <> \\
\quad > \\
\quad \text{attractions} = < \\
\quad \\
\quad ["la corrida"] = (\text{ATTRACTION}) <> \\
\quad ["Alcázar"] = (\text{HISTORIC\_SITE}) <> \\
\quad > \\
\quad > 
\]

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.

\[
\text{hotels} = (\text{List}<\text{HOTEL}>) < \\
\quad ["gran sevilla"] = (\text{HISTORIC\_HOTEL}) <> \\
\quad > 
\]

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 \text{org.openehr.rm.ehr.content ENTRY} and \text{Core\_Abstractions\_Relationships\_Relationship}.

\textbf{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.

### 3.4.6 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 association. This can be expressed as follows.

\[
\text{destinations} = < \\
\quad ["sofitel"] = (\text{LUXURY\_HOTEL}) <> \\
\quad ["hotel real"] = (\text{PENSION}) <> \\
\quad > \\
\quad \text{attractions} = < \\
\quad ["la corrida"] = (\text{ATTRACTION}) <> \\
\quad ["Alcázar"] = (\text{HISTORIC\_SITE}) <> \\
\quad > \\
\quad > 
\]
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.

### 3.4.6.1 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"]/
```

### 3.5 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.

\[
some\_attribute = \langle\text{manifest value here}\rangle
\]

### 3.5.1 Primitive Types

#### 3.5.1.1 Character Data

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

\`
'a'
\`

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

\`
\&ohgr;\ -- greek omega
\`

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

#### 3.5.1.2 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 \"phrase\".\"
```

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.\"\n```

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

```
\"a \&isin; A\" -- prints as: \(a \in A\)
```

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

#### 3.5.1.3 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 3.5.3 below).

#### 3.5.1.4 Real Data

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

```
25.0
```
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 3.5.3 below).

### 3.5.1.5 Boolean Data

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

- **True**
- **False**

### 3.5.1.6 Dates and Times

In dADL, full and partial dates, times and durations can be expressed. All full dates, times and durations are expressed in ISO8601 form. Patterns for dates and times based on ISO8601 include the following:

- (yyyy-MM-dd) -- a date
- (hh:mm:ss[.sss][Z]) -- a time with optional seconds
- (yyyy-MM-dd hh:mm:ss[.sss][Z]) -- a date/time

where:

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

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

- **1919-01-23** -- birthdate of Django Reinhardt
- **16:35:04.5** -- rise of Venus in Sydney on 24 Jul 2003
- **2001-05-12 07:35:20+1000** -- timestamp on an email received from Australia
- **P22D4H15M0S** -- period of 22 days, 4 hours, 15 minutes

Partial dates and times, i.e. dates and times with unknown parts are expressed in the same form but with the literal "??" for the unknown parts. **This is a deviation from ISO8601**, which specifies that partial dates and times can be expressed simply by omitting the unknown parts. However, this leads to strings like “12” (12 o'clock, minutes and seconds unknown), which cannot easily be recognised by parsers. Instead, dADL uses forms like “12:??:??” which are easily parsed as being dates and times. 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-dd hh:mm:??** -- date/time with unknown seconds
- **yyyy-MM-dd hh:??:??** -- date/time with unknown minutes and seconds
- **yyyy-MM-dd ???:??::??** -- date/time with unknown time
3.5.1.7 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 inclusive range where N and M are integers, or the infinity indicator;
\[|<N|\] less than N;
\[|>N|\] greater than N;
\[|>=N|\] greater than or equal to N;
\[|<=N|\] less than or equal to N;
\[|N +/-M|\] interval of N ± M.
\[!= N\] does not equal N;

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

- \[\text{infinity}\]
- \[-\text{infinity}\]
- * equivalent to \text{infinity}

Examples of this syntax include:

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

3.5.2 Other Built-in Types

3.5.2.1 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/rfc2396.txt. In the context of dADL, no quotes or inverted commas are needed; and neither spaces nor angle brackets are allowed; both have to be quoted e.g. %20 means a space (as per the URI standard). 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 Reserved Character Encoding on page 23.

3.5.2.2 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:

\[[\text{terminology_id}::\text{code}]\]

Typical examples from clinical data:

- [icd10am::F60.1] -- from ICD10AM
- [snomed-ct::2004950] -- from snomed-ct
3.5.3 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

3.6 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 serialised 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) <#
definition = (cadl) <#
definition = (cadl) <#
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.
3.7 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:

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

maps to the XML:

```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:

```plaintext
countries = [
  ["spain"] = <
    ["hotels"] = <...>,
    ["attractions"] = <...>
  >
]
```

This guarantees that the path `countries["spain"]/hotels/attractions` navigates to the same element in both dADL and the XML.
can be mapped to the XML in which the synthesised element tag “_items” and the attribute “key” are used:

```xml
<countries key="spain">
  <_items key="hotels">
    ...
  </_items>
  <_items key="attractions">
    ...
  </_items>
</countries>
<countries key="egypt">
  <_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:

```dADL
destinations = <
  [“seville”] = (<TOURIST_DESTINATION> <
    profile = (<DESTINATION_PROFILE>) <>
    hotels = <
      [“gran sevilla”] = (<HISTORIC_HOTEL>) <>
    >
  >
>`
```

maps to:

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

### 3.8 Syntax Specification

The dADL grammar is available as an [HTML document](http://svn.openehr.org/ref_impl_eiffel/TRUNK/libraries/common_libs/src/struc...). 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://svn.openehr.org/ref_impl_eiffel/TRUNK/libraries/common_libs/src/struc-](http://svn.openehr.org/ref_impl_eiffel/TRUNK/libraries/common_libs/src/struc-).
The Archetype Definition Language Version 2 (ADL2)

tures/syntax/dadl/parser/). The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

3.8.1 Grammar
The following provides the dADL parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository (http://svn.openehr.org/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
  | V_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 ]
```
dADL - Data ADL
The Archetype Definition Language Version 2 (ADL2)
Rev 2.0

single_attr_object_block:
   untyped_single_attr_object_block
   | V_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
   | V_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

string_value:
V_STRING

**string_list_value:**
V_STRING, V_STRING
| string_list_value, V_STRING
| V_STRING, SYM_LIST_CONTINUE

**integer_value:**
V_INTEGER
| + V_INTEGER
| - V_INTEGER

**integer_list_value:**
integer_value, integer_value
| integer_list_value, integer_value
| integer_value, SYM_LIST_CONTINUE

**integer_interval_value:**
SYM_INTERVAL_DELIM integer_value SYM_ELLIPSIS integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GT integer_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_GE integer_value SYM_INTERVAL_DELIM
| 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_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:
  precise_date_value
| partial_date_value

precise_date_value:
  V_INTEGER - V_INTEGER - V_INTEGER

partial_date_value:
  V_INTEGER - V_INTEGER - SYM_DT_UNKNOWN
| V_INTEGER - SYM_DT_UNKNOWN - SYM_DT_UNKNOWN

date_list_value:
  date_value, date_value
| date_list_value, date_value
| date_value, SYM_LIST_CONTINUE

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

precise_time_value:
  V_INTEGER : V_INTEGER : V_INTEGER
| V_INTEGER : V_INTEGER : V_REAL
| V_INTEGER : V_INTEGER

partial_time_value:
  V_INTEGER : V_INTEGER : SYM_DT_UNKNOWN
| V_INTEGER : SYM_DT_UNKNOWN : SYM_DT_UNKNOWN
| V_INTEGER : SYM_DT_UNKNOWN

time_zone:
  Z
| + V_INTEGER

time_list_value:
  time_value, time_value
| time_list_value, time_value
| time_value, SYM_LIST_CONTINUE

time_interval_value:
  SYM_INTERVAL_DELIM time_value SYM_ELLIPSIS time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LT time_value SYM_INTERVAL_DELIM
| SYM_INTERVAL_DELIM SYM_LE time_value SYM_INTERVAL_DELIM
The Archetype Definition Language Version 2 (ADL2)  
dADL - Data ADL  
Rev 2.0

| 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:  
  precise_date_time_value  
| partial_date_time_value  

precise_date_time_value:  
  precise_date_value precise_time_value  

partial_date_time_value:  
  precise_date_value partial_time_value  
  | precise_date_value SYM_DT_UNKNOWN : SYM_DT_UNKNOWN : SYM_DT_UNKNOWN  
  | partial_date_value SYM_DT_UNKNOWN : SYM_DT_UNKNOWN : SYM_DT_UNKNOWN  

date_time_list_value:  
  date_time_value , date_time_value  
  | date_time_list_value , date_time_value  
  | date_time_value , SYM_LIST_CONTINUE  

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

duration_value:  
  duration_magnitude  
  | - duration_magnitude  

duration_magnitude:  
  V_ISO8601_DURATION  

duration_list_value:  
  duration_value , duration_value  
  | duration_list_value , duration_value  
  | duration_value , SYM_LIST_CONTINUE  

duration_interval_value:  
  SYM_INTERVAL_DELIM duration_value SYM_ELLIPSIS duration_value SYM_INTERVAL_DELIM  
  | SYM_INTERVAL_DELIM SYM_LT duration_value SYM_INTERVAL_DELIM  
  | SYM_INTERVAL_DELIM SYM_LE duration_value SYM_INTERVAL_DELIM  
  | SYM_INTERVAL_DELIM SYM_GT duration_value SYM_INTERVAL_DELIM  
  | SYM_INTERVAL_DELIM SYM_GE duration_value SYM_INTERVAL_DELIM  
  | 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
3.8.2 Symbols
The following provides the dADL lexical analyser production rules (lex specification) as of revision 36 of the Eiffel reference implementation repository (http://svn.openehr.org/ref_impl_eiffel):

----------/** 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
3.9 Syntax Alternatives

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

3.9.1 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>
_hyper = <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 appropriate container types from a recognised standard such as OMG UML, OCL or IDL.
4 cADL - Constraint ADL

4.1 Overview

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

```
PERSON [at0000] matches { -- constraint on PERSON instance
    name matches { -- constraint on PERSON.name
        TEXT matches {/.*/} -- any non-empty string
    }
    addresses cardinality matches {0..*} matches { -- constraint on
        ADDRESS matches { -- PERSON.addresses
            -- etc --
        }
    }
}
```

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

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

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

In the standard cADL documented in this section, literal leaf values (such as the regular expression `/..*/` in the above example) are always constraints on a set of ‘standard’ widely-accepted primitive types, as described in the dADL section. Other more sophisticated constraint syntax types are described in cADL - Constraint ADL on page 47.
4.2 Basics

4.2.1 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
- include, exclude

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

<table>
<thead>
<tr>
<th>Textual Rendering</th>
<th>Symbolic Rendering</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches, is_in</td>
<td>∈</td>
<td>Set membership, “p is in P”</td>
</tr>
<tr>
<td>not, ~</td>
<td>~</td>
<td>Negation, “not p”</td>
</tr>
</tbody>
</table>

Keywords are shown in blue in this document.

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

aaa matches { /*ion[^s\n\t]*/ } -- the set of english words ending in ‘ion’

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

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

The meaning of the constraint structure above is: in data matching the constraints, there is an instance of type XXX whose attribute values recursively match the inner constraints named after those attributes, and so on, to the leaf level.

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}

1. was ‘use_archetype’, which is now deprecated
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`.

### 4.2.2 Comments

In a cADL 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.

### 4.2.3 Information Model Identifiers

As with dADL, identifiers from the underlying information model are used for all cADL nodes. Identifiers obey the same rules as in dADL: type names commence with an upper case letter, while attribute and function names commence with a lower case letter. In cADL, type names and any property (i.e. attribute or function) name can be used, whereas in dADL, only type names and attribute names appear.

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

Type 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. Identifiers are shown in green in this document.

### 4.2.4 Node Identifiers

In cADL, an entity in brackets e.g. `[xxxx]` is used to identify “object nodes”, i.e. nodes expressing constraints on instances of some type. Object nodes always commence with a type name. Any string may appear within the brackets, depending on how it is used. However, in this document, all node identifiers are of the form of an archetype term identifier, i.e. `[atNNNN]`, e.g. `[at0042]`. Node identifiers are shown in magenta in this document.

### 4.2.5 Natural Language

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

### 4.3 Structure

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

```c
PERSON[at0001] ∈ { aaa ~is_in {5}
aaa ⊈ {5}
```
In the above, any identifier (shown in green) followed by the ∈ operator (equivalent text keyword: matches or is_in) followed by an open brace, is the start of a “block”, which continues until the closing matching brace (normally visually indented to come under the start of the line at the beginning of the block).

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

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

![FIGURE 4 Object and Attribute Blocks in cADL](https://example.com/figure4.png)

### 4.3.1 Complex Objects

It may by now be clear that the identifiers in the above could correspond to entities in an object-oriented information model. A UML model compatible with the example above is shown in FIGURE 5. Note that there can easily be more than one model compatible with a given fragment of cADL syntax, and in particular, there may be more properties and classes in the reference model than are mentioned in the cADL constraints. In other words, a cADL text includes constraints only for those parts of a model that are useful or meaningful to constrain.

Constraints expressed in cADL cannot be stronger than those from the information model. For example, the PERSON.fam_name attribute is mandatory in the model in FIGURE 5, so it is not valid to
express a constraint allowing the attribute to be optional. In general, a cADL archetype can only further constrain an existing information model. However, it must be remembered that for very generic models consisting of only a few classes and a lot of optionality, this rule is not so much a limitation as a way of adding meaning to information. Thus, for a demographic information model which has only the types PARTY and PERSON, one can write cADL which defines the concepts of entities such as COMPANY, EMPLOYEE, PROFESSIONAL, and so on, in terms of constraints on the types available in the information model.

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

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

### 4.3.2 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>.

#### 4.3.2.1 Existence

The only constraint that applies to all attributes is to do with existence. Existence constraints say whether an attribute must exist, and are indicated by “0..1” or “1” markers at line ends in UML diagrams (and often mistakenly referred to as a “cardinality of 1..1”). It is the absence or presence of the cardinality constraint in cADL which indicates that the attribute being constrained is single-valued or a container attribute, respectively. Existence constraints are expressed in cADL as follows:

```cADL
QUANTITY matches {
    units existence matches 0..1 matches {"mm[Hg]"}
}
```
cADL - Constraint ADL

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

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

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

### 4.3.3 Single-valued Attributes

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

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

Here, the cardinality of the `value` attribute is 1..1 (the default), while the occurrences of both `QUANTITY` constraints is optional, leading to the result that only one `QUANTITY` instance can appear in runtime data, and it can match either of the constraints.

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

Note that there is a more efficient way to express the above example, using domain type extensions. An example is provided in section 10.1.2 on page 99.
4.3.4 Container Attributes

4.3.4.1 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 keyword *cardinality* indicates firstly that the property events must be of a container type, such as `List<T>`, `Set<T>`, `Bag<T>`. The integer range indicates the valid membership of the container; a single `∗` means the range `0..∗`, i.e. ‘0 to many’. The type of the container is not explicitly indicated, since it is usually defined by the information model. However, the semantics of a logical set (unique membership, ordering not significant), a logical list (ordered, non-unique membership) or a bag (unordered, non-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 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, what number of member items it must 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.*

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 --}
```

4.3.4.2 Occurrences

A constraint on occurrences is used only with cADL object nodes (not attribute nodes), to indicate how many times in runtime data an instance of a given class conforming to a particular constraint can occur. It only has significance for objects which are children of a container attribute, since by definition, the occurrences of an object which is the value of a single-valued attribute can only be `0..1` or `1..1`, and this is already defined by the attribute existence. However, it is not illegal. In the example below, three `EVENT` constraints are shown; the first one ("1 minute sample") is shown as mandatory, while the other two are optional.

```
events cardinality ∈ {∗} ∈ {
    EVENT[at0002] occurrences ∈ {0..1} ∈ {} -- 1 min sample
```

© 2003-2006 The openEHR Foundation

e-mail: info@openEHR.org web: http://www.openEHR.org

The Archetype Definition Language Version 2 (ADL2) cADL - Constraint ADL
Rev 2.0

Editors: {T Beale, S Heard}  Page 53 of 109  Date of Issue: 18 Jan 2006
EVENT[at0003] occurrences ∈ {0..1} ∈ {} -- 2 min sample
EVENT[at0004] occurrences ∈ {0..1} ∈ {} -- 3 min sample

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

GROUP[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 class may appear in data, each conforming to the one constraint block in the archetype. A single positive integer, or the infinity indicator, may also be used on its own, thus: {2}, {*}. The default occurrences, if none is mentioned, is {1..1}.

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

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

VCOC: cardinality/occurrences validity: the interval represented by: the sum of all occurrences minimum values - the sum of all occurrences maximum values must be inside the interval of the cardinality.

4.3.5 “Any” Constraints

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

PERSON[at0001] matches {
    name existence matches {0..1} matches {*}
    -- etc --
}

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

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

ELEMENT[at0004] matches {
    -- speed limit
}
value matches {
    QUANTITY matches {*}
}

The meaning of this constraint is that in the data at runtime, the value property of \texttt{ELEMENT} must be of type \texttt{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.

4.3.6 Object Node Identification and Paths

In many of the examples above, some of the object node typenames are followed by a node identifier, shown in brackets.

\textbf{Node identifiers} are required for any object node which is intended to be addressable elsewhere in the cADL text, or in the runtime system and which would otherwise be ambiguous i.e. has sibling nodes.

In the following example, the \texttt{PERSON} type does not require an identifier, since no sibling node exists at the same level, and unambiguous paths can be formed:

\begin{verbatim}
members cardinality ∈ {*} ∈ {
    PERSON ∈ {
        title ∈ ("head")
    }
}
\end{verbatim}

The path to the \textit{title} attribute is

\begin{verbatim}
members/title
\end{verbatim}

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

\begin{verbatim}
members cardinality ∈ {*} ∈ {
    PERSON[\texttt{at0104}] ∈ {
        title ∈ ("head")
    }
    PERSON[\texttt{at0105}] matches {
        title ∈ ("member")
    }
}
\end{verbatim}

The paths to the respective \textit{title} attributes are now:

\begin{verbatim}
members[\texttt{at0104}]/title
members[\texttt{at0105}]/title
\end{verbatim}

Logically, all non-unique parent nodes of an identified node must also be identified back to the root node. The primary function of node identifiers is in forming paths, enabling cADL nodes to be unambiguously referred to. The node identifier can also perform a second function, that of giving a design-time meaning to the node, by equating the node identifier to some description. Thus, in the example shown in section 4.3.1, the \texttt{ELEMENT} node is identified by the code \texttt{[at0010]}, which can be designated elsewhere in an archetype as meaning “diastolic blood pressure”.

Node ids are required only where it is necessary to create paths, for example in “use” statements. However, the underlying reference model might have stronger requirements. The openEHR EHR information models\cite{17} for example require that all node types which inherit from the class \texttt{LOCATA-}
BLE have both a archetype_node_id and a runtime name attribute. Only data types (such as QUANTITY, Coded_Text) and their constituent types are exempt.

Paths are used in cADL to refer to cADL nodes, and are expressed in the ADL path syntax, described in detail in section 7 on page 81. ADL paths have the same alternating object/attribute structure implied in the general hierarchical structure of cADL, obeying the pattern TYPE/attribute/TYPE/attribute/....

Paths in cADL always refer to object nodes, and can only be constructed through nodes having node ids, or nodes which are the only child object of a single-cardinality attribute.

Unusually for a path syntax, a trailing object identifier can be required, even if the attribute corresponds to a single relationship (as might be expected with the “name” property of an object) because in cADL, it is legal to define multiple alternative object constraints - each identified by a unique node id - for a relationship node which has single cardinality.

Consider the following cADL example:

```cADL
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
```

It is valid to add attribute references to the end of a path, if the underlying information model permits it, as in the following example.

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

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

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

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

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

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

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

An archetype internal reference is introduced with the use_node keyword, in a line of the following form:

use_node TYPE object_path

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

PERSON ∈ 
  identities ∈ 
    -- etc --
  }

  contacts cardinality ∈ {0..*} ∈ {
    CONTACT [at0002] ∈ 
      purpose ∈ {-- etc --}
      addresses ∈ {-- etc --}
    }
    CONTACT [at0003] ∈ 
      purpose ∈ {-- etc --}
      addresses ∈ {-- etc --}
    }
    CONTACT [at0004] ∈ 
      purpose ∈ {-- etc --}
      addresses cardinality ∈ {0..*} ∈ {
        ADDRESS [at0005] ∈ 
          type ∈ {-- etc --}
          details ∈ {-- etc --}
        ADDRESS [at0006] ∈ 
          type ∈ {-- etc --}
          details ∈ {-- etc --}
        ADDRESS [at0007] ∈ 
          type ∈ {-- etc --}
          details ∈ {-- etc --}
      }
    }
    CONTACT [at0008] ∈ 
      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
    }
  }

4.3.8 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”, 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 75). The slot might be “wide”, meaning it allows numerous other archetypes, or “narrow”, where it allows only a few or just one archetype. The point at where the slot occurs in the archetype is a “chaining point”, i.e. a point at which archetypes are chained together. An archetype slot is defined in terms of two lists of assertions statements defining which archetypes are allowed to and excluded from filling that slot. Since archetype slots are typed, the (possibly abstract) type of the allowed archetypes is already constrained. Otherwise, any assertion can be made. A basic kind of assertion is on the id of archetypes allowed in the slot, and it is possible to limit this to a single archetype, meaning that the choice of archetype in that slot is fixed. In general, however, the intention of archetypes is to provide highly re-usable models of real world content with local constraining left to templates. The full semantics of archetype chaining are described in the “Archetype Object Model” document.

An archetype slot is introduced with the keyword allow_archetype, and is expressed using two lists of assertions, each introduced with the keywords include and exclude, respectively.

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

```
SECTION [at2000] occurrences ∈ {0..1} ∈ {
    items ∈ {
        allow_archetype ENTRY occurrences ∈ {0..1} ∈ {
            include
                concept_short_name ∈ {/.+/}
        }
        allow_archetype SECTION occurrences ∈ {0..*} ∈ {
            include
                id ∈ {/.*\iso-ehr\section\.\.*\.*/}
            exclude
                id ∈ {/.*\iso-ehr\section\patient_details\.*/}
        }
    }
}
```

Here, every constraint inside the block starting on an allow_archetype line contains constraints that must be met by archetypes in order to fill the slot. (Note that in the examples above, and in all ADL, the ‘\’ character in paths is not a path separator, as some Microsoft Windows users might assume - it retains its meaning as a quoting character; above it is used to quote the ‘.’ character to ensure it has its literal meaning of ‘.’ rather than “any character” which is its regular expression meaning. Although not particularly beautiful, the syntax used above to indicate allowed values of the id attribute is completely standard regular expression syntax, and would be familiar to most users of Unix operating systems (e.g. Linux, BSD etc), Perl, and many other formalisms. Developers of archetypes using GUI tools should of course be spared such technical details.)

Other constraints are possible as well, including that the allowed archetype must contain a certain keyword, or a certain path. The latter is quite powerful – it allows archetypes to be linked together on the basis of context. For example, under a “genetic relatives” heading in a Family History Organiser archetype, the following logical constraint might be used:

```
allow_archetype ENTRY occurrences ∈ {0..*} matches {
    include
        short_concept_name ∈ {“family_history_subject”}
        \∧ ∃ /subject//relation
        \∧ /subject//.relation ∈ {
            CODED_TEXT ∈ {
```
4.3.9 Mixed Structures

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

- **complex object structures**: any node introduced by a type name and followed by {} containing constraints on attributes;
- **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.

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

```latex
SECTION[at2000] \in \{
\text{items cardinality} \in \{0..*; \text{ordered}\} \in \{
\text{ENTRY}[at2001] \in \{\text{etc}\},
\allowarchetype \text{ENTRY} \in \{\text{etc}\},
\usenode \text{ENTRY}[at0001]/\text{some_path}[at0004]/
\text{ENTRY}[at2002] \in \{\text{etc}\},
\usenode \text{ENTRY}/[at1002]/\text{some_path}[at1012]/
\usenode \text{ENTRY}/[at1005]/\text{some_path}[at1052]/
\text{ENTRY}[at2003] \in \{\text{etc}\}
\}\}
```

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

4.4 Constraints on Primitive Types

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

```latex
\text{some_attr matches}\{\text{some_pattern}\}
```

rather than:

```latex
\text{some_attr matches}\{
\text{BASIC_TYPE matches}\{
\text{some_pattern}
\}
\}
```

This is made possible because the syntax patterns of all primitive types 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. cADL does not however oblige the brief form described here, and the more verbose one can be used. In either case, the syntax of the pattern appearing within the final pair of braces obeys the rules described below.

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

4.4.1.1 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 trade-names (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 86.

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

The regular expression patterns supported in cADL are as follows.

**Atomic Items**
- match any single character. E.g. / . . . / matches any 3 characters which occur with a space before and after;
The Archetype Definition Language Version 2 (ADL2)  
cADL - Constraint ADL  
Rev 2.0

[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.

### 4.4.2 Constraints on Integer

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

#### 4.4.2.1 List of Integers

Lists of integers expressed in the syntax from dADL (described in Lists of Built-in Types on page 34) can be used as a constraint, e.g.:
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.

### 4.4.2.2 Interval of Integer

Integer intervals are expressed using the interval syntax from dADL (described in Intervals of Ordered Primitive Types on page 33). Examples include:

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

- 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 {<10|} -- allow up to 9
- length matches {10|} -- allow 10 or more
- length matches {10..110|} -- allow up to 10
- length matches {100+/-5|} -- allow 100 +/- 5, i.e. 95 - 105
- rate matches {0..infinity|} -- allow 0 - infinity, i.e. same as >= 0

Intervals may be combined in integer constraints, using the semicolon character (‘;’) as follows:

- critical_range matches {|5..9; 101..110|}

### 4.4.3 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} -- fixed value
- magnitude ∈ {5.5, 6.0} -- interval
- magnitude ∈ {<10.0|} -- allow anything less than 10.0
- magnitude ∈ {>=10.0|} -- allow 10.0 or more
- magnitude ∈ {|80.0+/-12.0|} -- allow 80 +/- 12

### 4.4.4 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}

### 4.4.5 Constraints on Character

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

#### 4.4.5.1 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'}

#### 4.4.5.2 Regular Expression

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

4.4.6 Constraints on Dates, Times and Durations

To Be Determined: sub-second precision not yet described

Dates, times, date/times and durations may all be constrained in three ways: using a list of strings, 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.

4.4.6.1 Patterns

Date, Time and Date/Time

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. The following table shows the valid patterns which can be used, and the types implied by each pattern. These are formed from the abstract pattern yyyy-mm-dd hh:mm:ss (itself formed by translating each field of an ISO 8601 date/time into a letter representing its type), with either ‘?’ (meaning optional) or ‘X’ (not allowed) characters substituted in appropriate places.

<table>
<thead>
<tr>
<th>Implied Type</th>
<th>Pattern</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>yyyy-mm-dd</td>
<td>full date must be specified</td>
</tr>
<tr>
<td>Date, Partial Date</td>
<td>yyyy-mm-??</td>
<td>optional day; e.g. day in month forgotten</td>
</tr>
<tr>
<td>Date, Partial Date</td>
<td>yyyy-??-??</td>
<td>optional month, day; i.e. any date allowed; e.g. mental health questionnaires which include well known historical dates</td>
</tr>
<tr>
<td>Partial Date</td>
<td>yyyy-??-XX</td>
<td>optional month, no day; (any examples?)</td>
</tr>
<tr>
<td>Time</td>
<td>hh:mm:ss</td>
<td>full time must be specified</td>
</tr>
<tr>
<td>Partial Time</td>
<td>hh:mm:XX</td>
<td>no seconds; e.g. appointment time</td>
</tr>
<tr>
<td>Partial Time</td>
<td>hh:??:XX</td>
<td>optional minutes, no seconds; e.g. normal clock times</td>
</tr>
<tr>
<td>Time, Partial Time</td>
<td>hh:??:??</td>
<td>optional minutes, seconds; i.e. any time allowed</td>
</tr>
<tr>
<td>Date/Time</td>
<td>yyyy-mm-dd hh:mm:ss</td>
<td>full date/time must be specified</td>
</tr>
<tr>
<td>Date/Time, Partial Date/Time</td>
<td>yyyy-mm-dd hh:mm:??</td>
<td>optional seconds; e.g. appointment date/time</td>
</tr>
</tbody>
</table>
Duration Constraints

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

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

### 4.4.6.2 List and Intervals of Dates, Times, etc

Dates, times, date/times and durations can also be constrained using lists and intervals. Each date, time etc in such a list or 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
- \(|\leq 09:30:00|\) -- any time at or before 9:30 am
- \(|> 09:30:00|\) -- any time after 9:30 am
- \(|\geq 09:30:00|\) -- any time at or after 9:30 am
- \(|2004-05-20..2004-06-02|\) -- a date range
- \(|2004-05-20 00:00:00..2005-05-19 23:59:59|\) -- a date/time range

### 4.4.7 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 for complex types), as follows:

\[
\text{some_attr \ cardinality} \in \{0..*\} \in \{\text{some_constraint}\}
\]

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,

\[
\text{speed_limits \ cardinality} \in \{0..*; \text{ordered}\} \in \{50, 60, 70, 80, 100, 130\}
\]

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.

### 4.4.8 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 contain an optional protocol section 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 optionally 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; 1800-01-01 00:00:00}`

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.

### 4.5 Syntax Specification

The cADL grammar is available as an [HTML document](http://my.openeHR.org/wsvn/ref_impl_eiffel/TRUNK/components/adl_parser/src/syntax/cadl/parser/?rev=0&sc=0). 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://svn.openehr.org/ref_impl_eiffel](http://svn.openehr.org/ref_impl_eiffel). The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

#### 4.5.1 Grammar

The following provides the cADL parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository (http://svn.openehr.org/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.

```plaintext
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:
   V_TYPE_IDENTIFIER | V_TYPE_IDENTIFIER V_LOCAL_TERM_CODE_REF

c_complex_object_body:
   c_any
```

© 2003-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
c_object:
    c_complex_object
    archetype_internal_ref
    archetype_slot
    constraint_ref
    c_coded_term
    c_ordinal
    c_primitive_object
    V_C_DOMAIN_TYPE
    ERR_C_DOMAIN_TYPE
    error

archetype_internal_ref:
    SYM_USE_NODE V_TYPE_IDENTIFIER object_path
    SYM_USE_NODE V_TYPE_IDENTIFIER error

archetype_slot:
    c_archetype_slot_head SYM_MATCHES SYM_START_CBLOCK c_includes c_excludes SYM_END_CBLOCK

    c_archetype_slot_head:
        c_archetype_slot_id c_occurrences

    c_archetype_slot_id:
        SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER
        SYM_ALLOW_ARCHETYPE V_TYPE_IDENTIFIER V_LOCAL_TERM_CODE_REF
        SYM_ALLOW_ARCHETYPE error

    c_primitive_object:
        c_primitive

        c_primitive:
            c_integer
            c_real
            c_date
            c_time
            c_date_time
            c_duration
            c_string
            c_boolean
            error

    c_any:
        *

    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
The Archetype Definition Language Version 2 (ADL2)
cADL - Constraint ADL
Rev 2.0

```
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_spec:
  V_ISO8601_DATE_CONSTRAINT_PATTERN
| date_value
| date_interval_value

c_date:
  c_date_spec
| c_date_spec ; date_value
| c_date_spec ; error

c_time_spec:
  V_ISO8601_TIME_CONSTRAINT_PATTERN
| time_value
| time_interval_value

c_time:
  c_time_spec
| c_time_spec ; time_value
| c_time_spec ; error

c_date_time_spec:
  V_ISO8601_DATE_TIME_CONSTRAINT_PATTERN
| date_time_value
| date_time_interval_value

c_date_time:
  c_date_time_spec
| c_date_time_spec ; date_time_value
| c_date_time_spec ; error

c_duration_spec:
  duration_value
| duration_interval_value

c_duration:
  c_duration_spec
| c_duration_spec ; duration_value
| c_duration_spec ; error

c_string_spec:
  V_STRING
| string_list_value
| string_list_value , SYM_LIST_CONTINUE
The Archetype Definition Language Version 2 (ADL2)  
cADL - Constraint ADL  
Rev 2.0

| \ 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

\c_ordinal:
  \ c_ordinal_spec
  | \ c_ordinal_spec ; \ integer_value
  | \ c_ordinal_spec ; \ error

\c_ordinal_spec:
  \ ordinal
  | \ c_ordinal_spec , \ ordinal

\ordinal:
  \ integer_value \ SYM_INTERVAL_DELIM \ V_QUALIFIED_TERM_CODE_REF

\c_coded_term:
  \ V_TERM_CODE_CONSTRAINT
  | \ V_QUALIFIED_TERM_CODE_REF

\constraint_ref:
  \ V_LOCAL_TERM_CODE_REF

\any_identifier:
  \ V_TYPE_IDENTIFIER
  | \ V_ATTRIBUTE_IDENTIFIER

-- for string_value etc, see dADL spec

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

-- for assertions, assertion, see Assertion spec

4.5.2 Symbols
The following shows the lexical specification for the cADL grammar.

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

----------/* symbols */ -------------------------------------------------
"-"  -- -> Minus_code
"+"  -- -> Plus_code

© 2003-2006 The openEHR Foundation  
email: info@openEHR.org web: http://www.openEHR.org
cADL - Constraint ADL
Rev 2.0

"*" -- -> 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][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] -- -> SYM_IMPLIES
[Tt][Rr][Uu][Ee] -- -> SYM_TRUE
[Ff][Aa][Ll][Ee] -- -> SYM_FALSE
[Ff][Oo][Rr][Aa][Ll] -- -> SYM_FORALL
[Ee][Xx][Ii][Ss][Tt][Tt][Ss] -- -> SYM_EXISTS
The Archetype Definition Language Version 2 (ADL2)

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

[Er][Xr][Ir][Sr][Tr][Er][Nr][Cr][Er] -- -> SYM_EXISTENCE
[Or][Cr][Cc][Ur][Rr][Er][Nr][Cr][Er][Sr] -- -> SYM_OCCURRENCES
[Cr][Ar][Dr][Ir][Nr][Ar][Lr][Ir][Tr][Yr] -- -> SYM_CARDINALITY
[Or][Rr][Dr][Er][Rr][Er][Dr] -- -> SYM_ORDERED
[Ur][Nr][Or][Rr][Dr][Er][Rr][Er][Dr] -- -> SYM_UNORDERED
[Ur][Nr][Ir][Qq][Ur][Er] -- -> SYM_UNIQUE
[Ir][Nr][Inf][Ir][Ir][Tr][Yr] -- -> SYM_INFINITY
[Ur][Sr][Er][_] [Nr][Or][Dr][Er] -- -> SYM_USE_NODE
[Ur][Sr][Ar][Cc][Hh][Ee][Tr][Yr][Pr][Er] -- -> SYM_ALLOW_ARCHETYPE
[Ar][Ll][Ar][Cc][Hh][Ee][Tr][Yr][Pr][Er] SYM_ALLOW_ARCHETYPE
[Ir][Nr][Cc][Lr][Ur][Dr][Er] -- -> SYM_INCLUDE
[Er][Xr][Cc][Lr][Ur][Dr][Er] -- -> SYM_EXCLUDE

--- /* V_URI */ -------------------------------

[a-z]+://[^> ]*

--- /* V_QUALIFIED_TERM_CODE_REF */ -------------------------------

-- any qualified code, e.g. [local::at0001], [local::ac0001], [loinc::700-0]

\[[a-zA-Z0-9()._\-]+::[a-zA-Z0-9._\-]+\]

--- /* 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

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

<IN_TERM_CONSTRAINT>[ \t]*[a-zA-Z0-9.._\-]+[ \t\n]* -- pick up \ line
<IN_TERM_CONSTRAINT>[ \t]*[a-zA-Z0-9.._\-]+[ \t\n]* -- pick up : line
<IN_TERM_CONSTRAINT>[ \t]*[a-zA-Z0-9.._\-]+[ \t\n]* -- do nothing
<IN_TERM_CONSTRAINT>[ \t]*[a-zA-Z0-9.._\-]+[ \t\n]* -- pick up ] line
---------/* V_LOCAL_TERM_CODE_REF */ ---------------------------------
-- any unqualified code, e.g. [at0001], [ac0001], [700-0]
\[a-zA-Z0-9-a-zA-Z0-9._-]*\]

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

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

---------/* V_ISO8601_DURATION */ --------------------------------------
P([0-9]+[dD]?([0-9]+[hH]?([0-9]+[mM]?([0-9]+[sS]?)?\]?)?

---------/* 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?]-

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

---------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
[A-Z][a-zA-Z0-9-9.]*

---------/* 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
-- containing "()" 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

---------/* V_REGEXP */ -------------------------------------

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

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

---------/* V_QUALIFIED_TERM_CODE_REF */ -------------------------------
\[a-zA-Z0-9-()._\-]+::[a-zA-Z0-9._\-]+\]

---------/* V_ISO8601_DURATION */ --------------------------------------
P([0-9]+[dD]?([0-9]+[hH]?([0-9]+[mM]?([0-9]+[sS]?)?\]?)?)?

---------/* 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?]-

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

---------/* V_ATTRIBUTE_IDENTIFIER */ ----------------------------------
[A-Z][a-zA-Z0-9-9.]*

---------/* 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
-- containing "()" 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

---------/* V_REGEXP */ -------------------------------------
The Archetype Definition Language Version 2 (ADL2) cADL - Constraint ADL
Rev 2.0

{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"]*" -- strings containing quotes, special characters etc
"[^\n"]*" -- beginning of a string
<IN_STR>\\ -- \ - append '\'
<IN_STR>\" -- " - append '"'
<IN_STR>&[a-zA-Z][a-zA-Z0-9_]*; -- match ISO special character
<IN_STR>&#x(\[a-fA-F0-9_]){4}; -- match W3C XML special character
<IN_STR>\[^\n"]*" -- match final end of string
<IN_STR>.|\n| -- Error
5 Assertions

5.1 Overview

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

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

The archetype assertion language is a small language of its own. Formally it is a first-order predicate logic with equality and comparison operators ( = , >, etc). It is very nearly a subset of the OMG’s emerging OCL (Object Constraint Language) syntax, and is very similar to the assertion syntax which has been used in the Object-Z [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). Further work will be done to more completely define the assertion language.

5.2 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.

<table>
<thead>
<tr>
<th>Textual Rendering</th>
<th>Symbolic Rendering</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches, is_in</td>
<td>∈</td>
<td>Set membership, “p is in P”</td>
</tr>
<tr>
<td>exists</td>
<td>∃</td>
<td>Existential quantifier, “there exists ...”</td>
</tr>
<tr>
<td>for_all</td>
<td>∀</td>
<td>Universal quantifier, “for all x...”</td>
</tr>
<tr>
<td>implies</td>
<td>⊃, →</td>
<td>Material implication, “p implies q”, or “if p then q”</td>
</tr>
<tr>
<td>and</td>
<td>∧</td>
<td>Logical conjunction, “p and q”</td>
</tr>
<tr>
<td>or</td>
<td>∨</td>
<td>Logical disjunction, “p or q”</td>
</tr>
<tr>
<td>xor</td>
<td>⊕</td>
<td>Exclusive or, “only one of p or q”</td>
</tr>
<tr>
<td>not, ~</td>
<td>¬, ~</td>
<td>Negation, “not p”</td>
</tr>
</tbody>
</table>

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

5.3 Operators

Assertion expressions can include arithmetic, relational and boolean operators, plus the existential and universal quantifiers.
5.3.1 Arithmetic Operators
The supported arithmetic operators are as follows:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Exponent: ^
- Modulo division: % - remainder after integer division

5.3.2 Equality Operators
The supported equality operators are as follows:

- Equality: =
- Inequality: <>

The semantics of these operators are of value comparison.

5.3.3 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. Their domain is limited to values of comparable types.

5.3.4 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.

5.3.5 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.
5.4 Operands

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

- **manifest constant**: any constant of any primitive type, expressed according to the dADL syntax for values
- **variable reference**: any name starting with ‘$’, e.g. $body_weight;
- **object reference**: a path referring to an object node, i.e. any path ending in a node identifier
- **property reference**: a path referring to a property, i.e. any path ending in “.property_name”

If an assertion is used in an archetype slot definition, its paths refer to the archetype filling the slot, not the one containing the slot.

5.5 Precedence and Parentheses

To Be Continued:

5.6 Syntax Specification

5.6.1 Grammar

The following provides the Assertion parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository (http://svn.openehr.org/ref_impl_eiffel). Note that because of interdependencies with ADL path and cADL production rules, practical implementations may have to include all production rules in one parser.

```plaintext
assertions:
    assertion
  | assertions assertion

assertion:
    any_identifier : Boolean_expression
  | Boolean_expression
  | any_identifier : error

Boolean_expression:
    SYM_EXISTS object_path
  | SYM_EXISTS error
  | ( Boolean_expression )
  | call_path
  | V_ATTRIBUTE_IDENTIFIER SYM_MATCHES SYM_START_CBLOCK c_primitive SYM_END_CBLOCK
  | SYM_TRUE
  | SYM_FALSE
  | SYM_NOT Boolean_expression
  | Arithmetic_expression = Arithmetic_expression
  | Arithmetic_expression SYM_NE Arithmetic_expression
  | Arithmetic_expression SYM_LT Arithmetic_expression
  | Arithmetic_expression SYM_GT Arithmetic_expression
  | Arithmetic_expression SYM_LE Arithmetic_expression
  | Arithmetic_expression SYM_GE Arithmetic_expression
  | Boolean_expression SYM_AND Boolean_expression
  | Boolean_expression SYM_OR Boolean_expression
  | Boolean_expression SYM_XOR Boolean_expression
  | Boolean_expression SYM_IMPLIES Boolean_expression
```
**Arithmetic_expression:**

- `call_path`
- `{ Arithmetic_expression }`
- `V_INTEGER`
- `V_REAL`
- `V_STRING`
- `V_CHARACTER`
- `+ Arithmetic_expression`
- `- Arithmetic_expression`
- `Arithmetic_expression + Arithmetic_expression`
- `Arithmetic_expression - Arithmetic_expression`
- `Arithmetic_expression * Arithmetic_expression`
- `Arithmetic_expression / Arithmetic_expression`
- `Arithmetic_expression ^ Arithmetic_expression`

### 5.6.2 Symbols

See the lexical specification for the cADL grammar.
6 Declarations

This section for a future release of ADL

To Be Determined: main problem of variables is that they must have names, which are language-dependent; imagine if there were a mixture of variables added by authors in different languages. The only solution is to name them with terms.

To Be Determined: Variables have to be treated as term coordinations, and should be coded e.g. using ccNNNN codes (“cc” = coordinated code). Then they can be given meanings in any language.

6.1 Predefined Variables

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

- $\textit{current\_date}$: \texttt{Date}; returns the date whenever the archetype is evaluated
- $\textit{current\_time}$: \texttt{Time}; returns time whenever the archetype is evaluated
- $\textit{current\_date\_time}$: \texttt{Date\_Time}; returns date/time whenever the archetype is evaluated

To Be Continued: these should be coded as well, using openEHR codes

6.2 Archetype-defined Variables

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

\begin{verbatim}
let $var\_name = reference
\end{verbatim}

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

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

\begin{verbatim}
invariant
   let $sys\_bp =
       /at0000[/data[at9001]/events[at9002]/data[at1000]/items[at1100]
   let $dia\_bp =
       /at0000[/data[at9001]/events[at9002]/data[at1000]/items[at1200]
   $sys\_bp >= $dia\_bp
\end{verbatim}

To Be Continued:
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:

\['/\]{attr_name \{object_id\} '/'}*

Essentially ADL paths consist of segments separated by slashes ('/'), where each segment is an attribute name with optional object identifier, indicated by brackets ('[]'). A path either finishes in a slash, and identifies an object node, or finishes in an attribute name, and identifies an attribute node.

ADL Paths are formed from an alternation of segments made up of an attribute name and optional object node identifier, 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[@node_id = ‘at0001’] -- the member of ‘items’ with key ‘at0001’
```
In the above, `meaning()` is a notional function is defined for Xpath in openEHR, 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.

### 7.3 Path Syntax

The ADL path grammar is available as an [HTML document](http://my.openehr.org/wsvn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path/?rev=0&sc=0). 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://my.openehr.org/wsvn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path/?rev=0&sc=0](http://my.openehr.org/wsvn/ref_impl_eiffel/libraries/common_libs/src/structures/object_graph/path/?rev=0&sc=0). The .l and .y files can easily be converted for use in another yacc/lex-based programming environment.

#### 7.3.1 Grammar

The following provides the ADL path parser production rules (yacc specification) as of revision 36 of the Eiffel reference implementation repository ([http://svn.openehr.org/ref_impl_eiffel](http://svn.openehr.org/ref_impl_eiffel)).

```
input:  
movable_path | location_path | error

location_path:  
  absolute_path | absolute_path / call_path_segment | relative_path | relative_path / call_path_segment

movable_path:  
  SYM_MOVABLE_LEADER 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

call_path_segment:  
  V_FEATURE_CALL_IDENTIFIER

7.3.2 Symbols

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

```
"." 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_TYPE_IDENTIFIER

[a-z][a-zA-Z0-9-]*[ ]\(\)  V_FEATURE_CALL_IDENTIFIER

[a-z][a-zA-Z0-9-]*  V_ATTRIBUTE_IDENTIFIER
8 ADL - Archetype Definition Language

This section describes ADL archetypes as a whole, adding a small amount of detail to the descriptions of dADL and cADL already given. The important topic of the relationship of the cADL-encoded definition section and the dADL-encoded ontology section is discussed in detail. In this section, only standard ADL (i.e. the cADL and dADL constructs and types described so far) is assumed. Archetypes for use in particular domains can also be built with more efficient syntax and domain-specific types, as described in Customising ADL on page 99, and the succeeding sections.

An ADL archetype is a dADL document with the structure shown below:

```xml
archetype_id = "some.archetype.id"
adl_version = "2.0"

is_controlled = True
parent_archetype_id = "some.other.archetype.id"
concept = [concept_code]
original_language = "lang"
translations = <

>  
description = <

>
definition = (cadl) #

   cADL plug-in section

#>

invariant = (aadl) #

   assertions plug-in section

#>

ontology = <

>

revision_history = <

>
```

In the above, all top-level attribute names are an exact match in name and type for a corresponding attribute in the openEHR Archetype Object Model (AOM). Optional parts are shown unbolded. In this document, top level attributes are usually described as ’sections’ of the archetype.

8.1 Basics

8.1.1 Order of Sections

As of ADL 2.0, order is no longer significant in archetypes. However, it is strongly recommended that the order above be respected, since it is the logical reading order, and some parser implementations may rely on it.

8.1.2 Keywords

ADL has no keywords of its own (i.e, distinct from the keywords in cADL and the invariant language), although depending on how parsers are built, the top-level attribute names might be regarded as special in some implementations.
8.1.3 Node Identification

In the definition section of an ADL archetype, a particular scheme of codes is used for node identifiers as well as for denoting constraints on textual (i.e. language dependent) items. Codes are either local to the archetype, or from an external lexicon. This means that the archetype description is the same in all languages, and is available in any language that the codes have been translated to. All term codes are shown in brackets ([ ]). Codes used as node identifiers and defined within the same archetype are prefixed with “at” and 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 the implied semantics of subsumption; additionally - the “dot” structuring is used as an optimisation on node identification.

8.1.4 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 8.5.4.

8.2 Header Sections

8.2.1 Archetype_id Section

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

\[ archetype_id = <"mayo.openehr-ehr-entry.haematology.v1" > \]

The multi-axial identifier identifies archetypes in a global space. The syntax of the identifier is described under Archetype Identification on page 15 in The openEHR Archetype System.

8.2.2 Adl_version Section

This mandatory section indicates the version of ADL being used in the archetype. Its value is one of the revision values of this specification, such as “2.0”; it typically looks as follows:

\[ adl_version = <"2.0" > \]

8.2.3 Is_controlled Section

This is an optional section indicating whether the archetype is change-controlled or not can be included after the version, as follows:

\[ is_controlled = <False> \]

The flag may have the values True and False only, and is an aid to software. Archetypes having is_controlled set to True must have the revision_history section included, while those with the value False, or no flag at all, may omit the revision_history section. This enables archetypes to be privately edited in an early development phase without generating large revision histories of little or no value.
8.2.4 Parent_archetype_id 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_id = <"mayo.openehr-ehr-entry.haematology-cbc.v1">
parent_archetype_id = <"mayo.openehr-ehr-entry.haematology.v1">
```

Here the identifier of the new archetype is derived from that of the parent by adding a new section to its domain concept section. See Archetype Identification on page 15 in The openEHR Archetype System.

8.2.5 Concept Section

All archetypes represent some real world concept, such as a “patient”, a “blood pressure”, or an “antenatal examination”. The concept section describes the overall concept using the code of a concept whose definition is given in the archetype ontology. Like any coded entity, it can be displayed in any language the archetype has been translated to. A typical concept section is as follows:

```
concept = <[@0010]> -- haematology result
```

In this concept definition, the term definition of [@0010] contains a more complete description corresponding to the notion implied by the short identifier “haematology-cbc” in the example archetype_id section shown above.

8.2.6 Original_language and Translations Sections

The original_language and translations sections include 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 due to translation. There can be only one original_language. The translations section is optional, but must be present if any translations are present in the description or ontology sections; if present, it must be updated every time a translation of the archetype is undertaken. The following shows a typical example.

```
original_language = <
<"en">
>
translations = <
["de"] = <
  author = <"Frederick Smith">
  accreditation = <"British Medical Translator id 00400595">
>
["ru"] = <
  author = <"Vladimir Korotkov">
  accreditation = <"Russion Translator id 892230A">
  other_details = <
    ["email"] = <"vladimir.korotkov@acme.translators.ru">
  >
>
>
```

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.
8.2.7 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.

```xml
<description>
  <original_author>
    <name>Dr J Joyce</name>
    <organisation>NT Health Service</organisation>
    <date>2003-08-03</date>
  </original_author>
  <lifecycle_state>initial</lifecycle_state>
  <archetype_package_uri>
  </archetype_package_uri>
  <details>
    <en>
      <purpose>archetype for diabetic patient review</purpose>
      <use>
        used for all hospital or clinic-based diabetic reviews, including first time. Optional sections are removed according to the particular review
      </use>
      <misuse>not appropriate for pre-diagnosis use</misuse>
      <original_resource_uri>
      </original_resource_uri>
    </en>
    <de>
      <purpose>Archetyp für die Untersuchung von Patienten mit Diabetes</purpose>
      <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</misuse>
      <original_resource_uri>
      </original_resource_uri>
    </de>
  </details>
</description>
```

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. The design of the objects in this section is specified in the AOM. 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 CEN MetaKnow standard, with inclusions from the HL7 Templates Proposal [5] and the meta-data of the SynEx and Australian GeHR archetypes.

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

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

```cADL
definition = (cadl) <#
  ENTRY[at0000] ∈ {
    name ∈ {
      CODED_TEXT ∈ {
        code ∈ {
          CODE_PHRASE ∈ {{ac0001}}
        }
      }
    }
  }
  data ∈ {
    HISTORY[at9001] ∈ {
      events cardinality ∈ {1..*} ∈ {
        EVENT[at9002] occurrences ∈ {0..1} ∈ {-- baseline
          name ∈ {
            CODED_TEXT ∈ {
              code ∈ {
                CODE_PHRASE ∈ {{ac0002}}
              }
            }
          }
        }
      }
    }
    data ∈ {
      ITEM_LIST[at1000] ∈ {
        items cardinality ∈ {2..*} ∈ {
          ELEMENT[at1100] ∈ {
            name ∈ {
              CODED_TEXT ∈ {
                code ∈ {
                  CODE_PHRASE ∈ {{ac0002}}
                }
              }
            }
          }
        }
      }
    }
  }
  value ∈ {
    QUANTITY ∈ {
      magnitude ∈ {{0..1000}}
      property ∈ {{properties::0944}}
        -- "pressure"
      units ∈ {{units::387}}
        -- "mm[Hg]"
    }
  }
  ELEMENT[at1200] ∈ {
    name ∈ {
      CODED_TEXT ∈ {
        code ∈ {
          CODE_PHRASE ∈ {{ac0003}}
        }
      }
    }
  }
  value ∈ {
    QUANTITY ∈ {
      magnitude ∈ {{0..1000}}
      property ∈ {{properties::0944}}
        -- "pressure"
    }
  }
#
```
This definition expresses constraints on instances of the types \texttt{ENTRY}, \texttt{HISTORY}, \texttt{EVENT}, \texttt{LIST\_S}, \texttt{ELEMENT}, \texttt{QUANTITY}, and \texttt{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 \texttt{[at9000] “any” node near the bottom}).

### 8.3.1 Design-time and Run-time paths

All non-unique sibling nodes in a cADL text which correspond to nodes in data which might be referred to from elsewhere in the archetype, or might be used for querying at runtime, require a node identifier, and it is usually preferable to assign a design-time meaning, enabling paths and queries to be expressed using logical meanings rather than meaningless identifiers. When data are created according to a cADL specification, the archetype node identifiers are written into the data, providing a reliable way of finding data nodes, regardless of what other runtime names might have been chosen by the user for the node in question. There are two reasons for 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 clearly shows the difference between design-time meanings associated with node ids and runtime names is the following, for the root node of a \texttt{SECTION} headings hierarchy representing the problem/SOAP headings (a simple heading structure commonly used by clinicians to record patient contacts under top-level headings corresponding to the patient’s problem(s), and under each problem heading, the headings “subjective”, “objective”, “assessment”, and “plan”).

\begin{verbatim}
SECTION[at0000] matches { -- problem
  name matches {
    CODED\_TEXT matches {
      code matches {[ac0001]}
    }
  }
}
\end{verbatim}

In the above, the node identifier \texttt{[at0000]} is assigned a meaning such as “clinical problem” in the archetype ontology section. The subsequent lines express a constraint on the runtime \texttt{name} attribute, using the internal code \texttt{[ac0001]}. The constraint \texttt{[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 \texttt{[acNNNN]} codes are described in Local Constraint Codes on page 86.

### 8.4 Invariant Section

The \texttt{Invariant} 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 \texttt{Definition} section. Any constraint which relates more than one property to another is in this cate-
gory, as are most constraints containing mathematical or logical formulae. Invariants are expressed in the archetype assertion language, described in section 5 on page 75.

An invariant statement is a first order predicate logic statement which can be evaluated to a boolean result at runtime. Objects and properties are referred to using paths.

The following simple example says that the speed in kilometres of some node is related to the speed-in-miles by a factor of 1.6:

```
invariant = (aadl) <#
  validity: /speed[at0002]/kilometres/magnitude = /speed[at0004]/miles/magnitude * 1.6
#>
```

Note that in a well-designed archetype, the ‘1.6’ above should be coded and included in the ontology section.

### 8.5 Ontology Section

#### 8.5.1 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 general layout of this section.

```
ontology = <
  terminologies_available = <“snomed_ct”, ...>
  term_definitions = <
    [“en”] = <
      [“at0000”] = <...>
    ...
    [“de”] = <
      [“at0000”] = <...>
    ...
  >
  term_binding = <
    [“snomed_ct”] = <
      [“at0000”] = <...>
    ...
  >
  constraint_definitions = <
    [“en”] = <...>
      [“ac0001”] = <...>
    ...
    [“de”] = <
      [“ac0001”] = <...>
    ...
  >
  constraint_binding = <
    [“snomed_ct”] = <...>
```
The term_definitions section is mandatory, and must be defined for each translation carried out. Each of these sections can have its own meta-data, which appears within description sub-sections, such as the one shown above providing translation details.

### 8.5.2 Terminologies_available Sub-section

This section provides the total list of external terminologies which are mentioned anywhere else in the ontology.

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

### 8.5.3 Term_definitions Sub-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 must at 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"] = <
    ["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"] = <
    ["at0000"] = <
      text = "klinisches Problem"
      description = "Das Problem des Patienten worauf sich diese Informationen beziehen"
    >
    ["at0001"] = <
      text = "Problem/SOAP Schema"
      description = "SOAP-Schlagwort-Gruppierungschema fuer mehrfache Probleme"
    >
    ["at4000"] = <
      text = "Plan"
      description = "Klinisch-professionelle Beratung des Pflegenden"
    >
  >
```

In some cases, term definitions may have been lifted from existing terminologies. This is only a safe thing to do if the following conditions apply:

- if there is a lexical match between the intended term in the archetype and the preferred term or a synonym in the terminology;
- the definitions exactly match the need in the archetype;
- if the classification of the term in the terminology would classify real world instances (e.g. people with a certain illness) the same way as intended by the archetype design.

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 8.5.5 and section 8.5.6.

### 8.5.4 Constraint_definitions Sub-section

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

```
["at1015"] = <
  text = "type of hepatitis"
  description = "any term which means a kind of viral hepatitis"
>
```

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

### 8.5.5 Term_binding Sub-section

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

A typical term binding section resembles the following:

```
term_binding = <
  ["umls"] = <
    ["at0000"] = <[umls::C124305]> -- apgar result
    ["at0002"] = <[umls::00000000]> -- 1-minute event
  >
```

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

8.5.6 Constraint_binding Sub-section

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

```daml
constraint_binding = <
  ["snomed_ct"] = <
    ["ac0001"] = <http://terminology.org?terminology_id=snomed_ct&&
      has_relation=[102002];with_target=[128004] >
    ["ac0002"] = <http://terminology.org?terminology_id=snomed_ct&&
      synonym_of=[128025] >
  >
>
```

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

8.6 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 an equivalent 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).

```daml
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">
```
8.7 Validity Rules

This section describes the formal (i.e. checkable) semantics of ADL archetypes. It is recommended that parsing tools use the identifiers published here in their error messages, as an aid to archetype designers.

8.7.1 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 openEHR 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 cADL syntax string, or in an equivalent plug-in syntax.

- **VARDT**: archetype definition typename validity. The topmost typename mentioned in the archetype definition section must match the type mentioned in the type-name slot of the first segment of the archetype id.

8.7.2 Coded Term Validity

All node identifiers (‘at’ codes) used in the definition part of the archetype must be defined in the term_definitions part of the ontology.

- **VATDF**: archetype term validity. Each archetype term used as a node identifier the archetype definition must be defined in the term_definitions part of the ontology.

All constraint identifiers (‘ac’ codes) used in the definition part of the archetype must be defined in the constraint_definitions part of the ontology.

- **VACDF**: node identifier validity. Each constraint code used in the archetype definition part must be defined in the constraint_definitions part of the ontology.
8.7.3 Definition Section

The following constraints apply to the definition section of the archetype.

VDFAI: archetype identifier validity in definition. Any archetype identifier mentioned in an archetype slot in the definition section must conform to the published openEHR specification for archetype identifiers.

VDFPT: path validity in definition. Any path mentioned in the definition section must be valid syntactically, and a valid path with respect to the hierarchical structure of the definition section.
9 The ADL Parsing Process

9.1 Overview

FIGURE 6 illustrates the ADL parsing process. An ADL file is converted by the ADL parser into an ADL parse tree. This tree is an in-memory object structure representation of the semantics of the archetype, in a form corresponding to the openEHR Archetype Object Model. This model is then validated by the semantic checker of the ADL parser, which can verify numerous things, such as that term codes referenced in the definition section are defined in the ontology section. It can also validate the classes and attributes mentioned in the archetype against a specification for the relevant information model (e.g. in XMI or some equivalent).

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

- **C_complex_object**: any interior node representing a constraint on instances of some non-primitive type, e.g. ENTRY, SECTION;
- **C_attribute**: a node representing a constraint on an attribute (i.e. UML ‘relationship’ or ‘primitive attribute’) in an object type;

**FIGURE 6** Parsed ADL Structure
**C_primitive_object**: a node representing a constraint on a primitive (built-in) object type;

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

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

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

See the openEHR Archetype Object Model (AOM) for details.
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 which is isomorphic to the structure of the corresponding classes in the reference model; constraints at the leaf nodes are represented in a syntactic way which avoids committing to particular modelling details. The overall result enables constraints on most reference model types to be expressed.

10.1.1 Custom Syntax

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

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

10.1.2 Custom Constraint Classes

Another situation in which standard ADL falls short is when the required semantics of constraint are different from those provided by the standard approach. Consider a simple type QUANTITY, shown at the top of FIGURE 8, which could be used to represent a person’s age in data. A typical ADL constraint to enable QUANTITY to be used to represent age in clinical data is shown below, followed by its
expression in ADL. The only way to do this in ADL is to use multiple alternatives. While this is a perfectly legal approach, it makes processing by software difficult, since the way such a constraint would be displayed in a GUI would be factored differently.

```
QUANTITY
  property: String
  magnitude: Real
  units: String
```

A more powerful possibility is to introduce a new class into the archetype model, representing the concept “constraint on QUANTITY”, which we will call C_QUANTITY here. Such a class fits into the class model of archetypes (described in the openEHR Archetype Model document), inheriting from the class C_DOMAIN_TYPE. The C_QUANTITY class is illustrated in FIGURE 9, and corresponds to the way constraints on QUANTITY objects are expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

```
age matches {
  QUANTITY matches {
    property matches "time"
    units matches "years" or "months"
    if units is "years" then magnitude matches 0..200
    if units is "months" then magnitude matches 3..36
  }
  QUANTITY matches {
    property matches "time"
    units matches "months"
    magnitude matches [3.0..12.0]
  }
  ...
}
```

**FIGURE 8** Standard ADL for Constraint on a Quantity Class

The question now is how to express a constraint corresponding to this class in an ADL archetype. The solution is logical, and uses standard ADL. Consider that a particular constraint on a QUANTITY must

```
C_QUANTITY
  property: String
  list
    C_QUANTITY_ITEM
      units[1]: String
      magnitude[0..1]: Interval<Real>
```

**FIGURE 9** Custom Constraint Type for QUANTITY
be an instance of a `C_QUANTITY`; which can be expressed at the appropriate point in the archetype in the form of a section of dADL - the data syntax used in the archetype ontology.

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

**FIGURE 10** Inclusion of a Constraint Object as Data

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

- the dADL section occurs inside the `{}` block where its standard ADL equivalent would have occurred (i.e. no other delimiters or special marks are needed);
- the dADL section must be ‘typed’, i.e. it must start with a type name, which should be a rule-based transform of a reference model type (as described in Adding Type Information on page 28);
- the dADL instance must obey the semantics of the custom type of which it is an instance.

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

11.1 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.

11.2 Constraint Syntaxes

11.2.1 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.

11.3 Ontology Formalisms

11.3.1 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
Relationship of ADL to Other Formalisms

The Archetype Definition Language Version 2 (ADL2)

Rev 2.0

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.
Lastly, OWL (as of mid 2004) is still under development, and has only a very limited set of primitive constraint types (it is not possible for example to state a constraint on an Integer attribute of the form ‘any value between 80 and 110’), although this is being addressed; by contrast, ADL provides a rich set of constraints on primitive types, including dates and times.

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).

### 11.3.2 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:

```lisp
(defunrelation 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].

11.4 XML-based Formalisms

11.4.1 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 openEHR 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.
A References

Publications


Resources


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


END OF DOCUMENT
The Archetype Object Model

Editor: T Beale

Contributors: {A Goodchild, Z Tun}, {T Austin, D Kalra, N Lea, D Lloyd}

Revision: 2.0

Pages: 53

Keywords: EHR, health records, constraints, archetypes

© 2004-2006 The openEHR Foundation

The openEHR foundation is an independent, non-profit community, facilitating the creation and sharing of health records by consumers and clinicians via open-source, standards-based implementations.

Founding Chairman

Founding Members

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
# Amendment Record

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Raiser</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5.1</td>
<td>Corrected documentation error - return type of ARCHETYPE_CONSTRAINT.has_path; add optionality markers to Primitive types UML diagram. Removed erroneous aggregation marker from ARCHETYPE_ONTOLOGY.parent_archetype and ARCHETYPE_DESCRIPTION.parent_archetype.</td>
<td>D Lloyd</td>
<td>20 Jan 2005</td>
</tr>
<tr>
<td>0.5</td>
<td>CR-000110. Update ADL document and create AOM document. Includes detailed input and review from: - DSTC - CHIME, University College London - Ocean Informatics Initial Writing. Taken from ADL document 1.2draft B.</td>
<td>T Beale</td>
<td>10 Nov 2004</td>
</tr>
</tbody>
</table>
Trademarks

Microsoft is a trademark of the Microsoft Corporation
# Introduction

1.1 Purpose ................................................................. 7  
1.2 Related Documents .................................................. 7  
1.3 Nomenclature ........................................................... 7  
1.4 Status ................................................................. 7  
1.5 Background ........................................................... 7  
1.5.1 What is an Archetype? ........................................ 7  
1.5.2 Context ............................................................ 8  
1.6 Tools ................................................................. 8  
1.7 Changes from Previous Versions ................................. 8  
1.7.1 Version 0.6 to 2.0 .............................................. 8  

# The Archetype Object Model

2.1 Design Background .................................................. 9  
2.2 Package Structure ..................................................... 9  
2.3 Model Overview ....................................................... 10  
2.3.1 Archetypes as Objects ....................................... 10  
2.3.2 The Archetype Ontology ..................................... 12  
2.3.3 Archetype Specialisation .................................... 12  
2.3.4 Archetype Composition ...................................... 12  
2.4 The Archetype Package ............................................. 13  
2.4.1 Overview ......................................................... 13  
2.4.2 ARCHETYPE Class ............................................. 14  
2.4.3 VALIDITY_KIND Class ......................................... 16  
2.5 Constraint Model Package ....................................... 17  
2.5.1 Overview ......................................................... 17  
2.5.2 Semantics ........................................................ 17  
2.5.3 ARCHETYPE_CONSTRAINT Class ....................... 22  
2.5.4 C_ATTRIBUTE Class .......................................... 22  
2.5.5 C_SINGLE_ATTRIBUTE Class ............................. 23  
2.5.6 C_MULTIPLE_ATTRIBUTE Class ......................... 23  
2.5.7 CARDINALITY Class .......................................... 24  
2.5.8 C_OBJECT Class ................................................ 25  
2.5.9 C_COMPLEX_OBJECT Class ................................. 25  
2.5.10 ARCHETYPE_SLOT Class ................................... 26  
2.5.11 ARCHETYPE_INTERNAL_REF Class ..................... 26  
2.5.12 CONSTRAINT_REF Class ................................... 26  
2.5.13 C_PRIMITIVE_OBJECT Class ......................... 27  
2.5.14 C_DOMAIN_TYPE Class .................................... 28  
2.6 The Assertion Package ............................................ 29  
2.6.1 Overview ......................................................... 29  
2.6.2 Semantics ........................................................ 29  
2.6.3 ASSERTION Class .............................................. 30  
2.6.4 EXPR_ITEM Class ............................................. 30  
2.6.5 EXPR_LEAF Class .............................................. 31  
2.6.6 EXPR_OPERATOR Class ..................................... 31  
2.6.7 EXPR_UNARY_OPERATOR Class ......................... 31  
2.6.8 EXPR_BINARY_OPERATOR Class ....................... 32  
2.6.9 OPERATOR_KIND Class ..................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7</td>
<td>The Primitive Package</td>
<td>35</td>
</tr>
<tr>
<td>2.7.1</td>
<td>C_PRIMITIVE Class</td>
<td>36</td>
</tr>
<tr>
<td>2.7.2</td>
<td>C_BOOLEAN Class</td>
<td>36</td>
</tr>
<tr>
<td>2.7.3</td>
<td>C_STRING Class</td>
<td>37</td>
</tr>
<tr>
<td>2.7.4</td>
<td>C_INTEGER Class</td>
<td>37</td>
</tr>
<tr>
<td>2.7.5</td>
<td>C_REAL Class</td>
<td>38</td>
</tr>
<tr>
<td>2.7.6</td>
<td>C_DATE Class</td>
<td>38</td>
</tr>
<tr>
<td>2.7.7</td>
<td>C_TIME Class</td>
<td>39</td>
</tr>
<tr>
<td>2.7.8</td>
<td>C_DATE_TIME Class</td>
<td>40</td>
</tr>
<tr>
<td>2.7.9</td>
<td>C_DURATION Class</td>
<td>42</td>
</tr>
<tr>
<td>2.8</td>
<td>Ontology Package</td>
<td>43</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Overview</td>
<td>43</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Semantics</td>
<td>43</td>
</tr>
<tr>
<td>2.8.3</td>
<td>ARCHETYPE_ONTOLOGY Class</td>
<td>44</td>
</tr>
<tr>
<td>2.8.4</td>
<td>ARCHETYPE_TERM Class</td>
<td>46</td>
</tr>
<tr>
<td>A</td>
<td>Domain-specific Extension Example</td>
<td>47</td>
</tr>
<tr>
<td>A.1</td>
<td>Overview</td>
<td>47</td>
</tr>
<tr>
<td>A.2</td>
<td>Scientific/Clinical Computing Types</td>
<td>47</td>
</tr>
<tr>
<td>B</td>
<td>Using Archetypes with Diverse Reference Models</td>
<td>49</td>
</tr>
<tr>
<td>B.1</td>
<td>Overview</td>
<td>49</td>
</tr>
<tr>
<td>B.2</td>
<td>Clinical Computing Use</td>
<td>49</td>
</tr>
<tr>
<td>C</td>
<td>References</td>
<td>51</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Purpose

This document describes a generic object model for archetypes, based only upon the generally accepted semantics of object models (typified by the OMG UML meta-model). The model presented here can be used as a basis for building software that processes archetypes, independent of their persistent representation; equally, it can be used to develop the output side of parsers that process archetypes in a linguistic format, such as the openEHR Archetype Definition Language (ADL) [4], XML-instance and so on. As a specification, it can be treated as an API for archetypes.

It is recommended that the openEHR ADL document [4] be read in conjunction with this document, since it contains a detailed explanation of the semantics of archetypes, and many of the examples are more obvious in ADL, regardless of whether ADL is actually used with the object model presented here or not.

1.2 Related Documents

Related documents include:

- The openEHR Archetype Definition Language (ADL)
- The openEHR Archetype Profile (oAP)

1.3 Nomenclature

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

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://svn.openehr.org/specification/TAGS/Release-1.0/publishing/architecture/am/aom.pdf.

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

Blue text indicates sections under active development.

1.5 Background

1.5.1 What is an Archetype?

Archetypes are constraint-based models of domain entities, or what some might call “structured business rules”. Each archetype describes configurations of data instances whose classes are described in a reference model; the instance configurations are considered to be valid exemplars of a particular domain concept. Thus, in medicine, an archetype might be designed to constrain configurations of instances of a simple node/arc information model, that express a “microbiology test result” or a “physical examination”. Archetypes can be composed, specialised, and templated for local use. The
The archetype concept has been described in detail by Beale [1], [2]. Most of the detailed formal semantics are described in the openEHR Archetype Definition Language [4]. The openEHR archetype framework is described in terms of Archetype Definitions and Principles [4] and an Archetype System [5].

1.5.2 Context
The object model described in this document relates to linguistic forms of archetypes as shown in FIGURE 1. The model (upper right in the figure) is the object-oriented semantic equivalent of the ADL the Archetype Definition Language BNF language definition, and, by extension, any formal transformation of it. Instances of the model (lower right on the figure) are themselves archetypes, and correspond one-to-one with archetype documents expressed in ADL or a related language.

1.6 Tools
Various tools exist for creating and processing archetypes. The openEHR tools are available in source and binary form from the website (http://www.openEHR.org).

1.7 Changes from Previous Versions

1.7.1 Version 0.6 to 2.0
As part of the changes carried out to ADL version 1.3, the archetype object model specified here is revised, also to version 2.0, to indicate that ADL and the AOM can be regarded as 100% synchronised specifications.

- added a new attribute `adl_version`: String to the `ARCHETYPE` class;
- changed name of `ARCHETYPE.concept_code` attribute to `concept`. 
2 The Archetype Object Model

2.1 Design Background

An underpinning principle of openEHR is the use of archetypes and templates, which are formal models of domain concepts controlling data structure and content of data. The elements of this architecture are twofold.

- The openEHR Reference Model (RM), defining the structure and semantics of information, and the service models (SMs), describing service interfaces. These models correspond respectively to the ISP RM/ODP information and computational viewpoints. The information models define the data of openEHR EHR systems; meaning that every data instance in a system is an instance of a type defined in the Information Model (or to be completely correct, the corresponding type in the relevant ITS). The information model is designed to be invariant in the long term, to minimise the need for software and schema updates.

- The openEHR Archetype Model (AM), defining the structure and semantics of archetypes and templates. The AM consists of the archetype language definition language (ADL), the Archetype Object Model (AOM), the openEHR Archetype profile (OAP) and the Template Object Model (TOM).

The purpose of ADL is to provide an abstract syntax for textually expressing archetypes and templates. The AOM defines the object model equivalent, in terms of a UML model. It is a generic model, meaning that it can be used to express archetypes for any reference model in a standard way. ADL and the AOM are brought together in an ADL parser: a tool which can read ADL archetype texts, and whose parse-tree (resulting in-memory object representation) is instances of the AOM. The TOM defines the object model of templates, which are themselves used to put archetypes together into local information structures, usually corresponding to screen forms.

The purpose of the openEHR Archetype Profile is to define which classes and attributes of the openEHR RM can sensibly archetyped, and to provide custom archetype classes.

2.2 Package Structure

The openEHR Archetype Object Model is defined as the package am.archetype, as illustrated in FIGURE 2. It is shown in the context of the openEHR am.archetype packages.

![FIGURE 2 openehr.am.archetype Package](image-url)
2.3 Model Overview

The model described here is a pure object-oriented model that can be used with archetype parsers and software that manipulates archetypes. It is independent of any particular linguistic expression of an archetype, such as ADL or OWL, and can therefore be used with any kind of parser. It is dependent only on two groups of assumed types. The first group includes the following primitive inbuilt types, whose names and assumed semantics are described by ISO 11404 (the exact openEHR correspondences are described in the openEHR Support Information Model).

- Boolean
- Character
- Integer

The second are assumed library types:

- String
- Date
- Time
- Date_time
- Duration
- Hash <T, K:Comparable> (keyed list of items of any type)
- Interval <T:Comparable> (interval of instances of any ordered type)

These types are supported in most implementation technologies, including XML, Java and other programming languages. They are not defined in this specification, allowing them to be mapped to the most appropriate concrete types in each implementation technology.

The openEHR types used are:

- ARCHETYPE_ID
- HIER_OBJECT_ID
- TERMINOLOGY_ID
- CODE_PHRASE
- DV_CODED_TEXT

The last of these can be found in the openEHR RM.Data_types.Text package, and is used only to represent coded term values in the model. The remaining *_ID types can be found in the openEHR RM.Common.Identification package.

2.3.1 Archetypes as Objects

FIGURE 3 illustrates various processes that can be responsible for creating an archetype object structure, including parsing, database retrieval and GUI editing. A parsing process that would typically turn a syntax expression of an archetype (ADL, XML, OWL) into an object one. The input file is converted by a parser into an object parse tree, shown on the right of the figure, whose types are specified in this document. Database retrieval will cause the reconstruction of an archetype in memory from a structured data representation, such as relational data, object data or XML. Direct in-memory editing by a user with a GUI archetype editor application will cause on-the-fly creation and destruction of parts of an archetype during the editing session, which would eventually cause the archetype to be stored in some form when the user decides to commit it.

As shown in the figure, the definition part of the in-memory archetype consists of alternate layers of object and attribute constrainer nodes, each containing the next level of nodes. In this document, the word ‘attribute’ refers to any data property of a class, regardless of whether regarded as a ‘relationship’ (i.e. association, aggregation, or composition) or ‘primitive’ (i.e. value) attribute in an object.
The Archetype Object Model

model. At the leaves are primitive object constrainer nodes constraining primitive types such as String, Integer etc. There are also nodes that represent internal references to other nodes, constraint reference nodes that refer to a text constraint in the constraint binding part of the archetype ontology, and archetype constraint nodes, which represent constraints on other archetypes allowed to appear at a given point. The full list of node types is as follows:

- **C_complex_object**: any interior node representing a constraint on instances of some non-primitive type, e.g. ENTRY, SECTION;
- **C_attribute**: a node representing a constraint on an attribute (i.e. UML ‘relationship’ or ‘primitive attribute’) in an object type;
- **C_primitive_object**: an node representing a constraint on a primitive (built-in) object type;
- **Archetype_internal_ref**: a node that refers to a previously defined object node in the same archetype. The reference is made using a path;
- **Constraint_ref**: a node that refers to a constraint on (usually) a text or coded term entity, which appears in the ontology section of the archetype, and in ADL, is referred to with an “acNNNN” code. The constraint is expressed in terms of a query on an external entity, usually a terminology or ontology;
- **Archetype_slot**: a node whose statements define a constraint that determines which other archetypes can appear at that point in the current archetype. It can be thought of like a keyhole, into which few or many keys might fit, depending on how specific its shape is. Logically it has the same semantics as a C_COMPLEX_OBJECT, except that the constraints are expressed in another archetype, not the current one.
The Archetype Object Model
Rev 2.0

The Archetype Object Model
Rev 2.0

The typenname nomenclature “C_complex_object”, “C_primitive_object”, “C_attribute” used here is intended to be read as “constraint on xxxx”, i.e. a “C_complex_object” is a “constraint on a complex object (defined by a complex reference model type)”. These typenames are used below in the formal model.

2.3.2 The Archetype Ontology

There are no linguistic entities at all in the definition part of an archetype, with the possible exception of constraints on text items which might have been defined in terms of regular expression patterns or fixed strings. All linguistic entities are defined in the ontology part of the archetype, in such a way as to allow them to be translated into other languages in convenient blocks. As described in the openEHR ADL document, there are four major parts in an archetype ontology: term definitions, constraint definitions, term bindings and constraint bindings. The former two define the meanings of various terms and textual constraints which occur in the archetype; they are indexed with unique identifiers which are used within the archetype definition body. The latter two ontology sections describe the mappings of terms used internally to external terminologies. Due to the well-known problems with terminologies (described in some detail in the openEHR ADL document, and also by e.g. Rector [6] and others), mappings may be partial, incomplete, approximate, and occasionally, exact.

2.3.3 Archetype Specialisation

Archetypes can be specialised. The formal rules of specialisation are described in the openEHR Archetype Semantics document (forthcoming), but in essence are easy to understand. Briefly, an archetype is considered a specialisation of another archetype if it mentions that archetype as its parent, and only makes changes to its definition such that its constraints are ‘narrower’ than those of the parent. Any data created via the use of the specialised archetype is thus conformant both to it and its parent. This notion of specialisation corresponds to the idea of ‘substitutibility’, applied to data.

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

2.3.4 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 a multitude of others. Archetype slots are the means of composition, and are themselves defined in terms of constraints.
2.4 The Archetype Package

2.4.1 Overview

The model of an archetype, illustrated in FIGURE 4, is straightforward at an abstract level, mimicking the structure of an archetype document as defined in the openEHR Archetype Definition Language (ADL) document. An archetype is modelled as a particular kind of AUTHORED_RESOURCE, and as such, includes descriptive meta-data, language information and revision history. The ARCHETYPE class adds identifying information, a definition - expressed in terms of constraints on instances of an object model, and an ontology. The archetype definition, the ‘main’ part of an archetype, is an instance of a C_COMPLEX_OBJECT, which is to say, the root of the constraint structure of an archetype always takes the form of a constraint on a non-primitive object type. The last section of an archetype, the ontology, is represented by its own class, and is what allows the archetypes to be natural language- and terminology-neutral.

A utility class, VALIDITY_KIND is also included in the Archetype package. This class contains one integer attribute and three constant definitions, and is intended to be used as the type of any attribute in this constraint model whose value is logically ‘mandatory’, ‘optional’, or ‘disallowed’. It is used in this model in the classes C_Date, C_Time and C_Date_Time.
### 2.4.2 ARCHETYPE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Archetype equivalent to ARCHETYPED class in Common reference model. Defines semantics of identification, lifecycle, versioning, composition and specialisation.</td>
</tr>
<tr>
<td>Inherit</td>
<td>AUTHORED_RESOURCE</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>archetype_id: ARCHETYPE_ID</td>
<td></td>
</tr>
<tr>
<td>uid: HIER_OBJECT_ID</td>
<td></td>
</tr>
<tr>
<td>concept_code: String</td>
<td></td>
</tr>
<tr>
<td>parent_archetype_id: ARCHETYPE_ID</td>
<td></td>
</tr>
<tr>
<td>definition: C_COMPLEX_OBJECT</td>
<td></td>
</tr>
<tr>
<td>ontology: ARCHETYPE_ONTOLOGY</td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td>Signature</td>
</tr>
<tr>
<td>version: String</td>
<td></td>
</tr>
<tr>
<td>previous_version: String</td>
<td></td>
</tr>
<tr>
<td>short_concept_name: String</td>
<td></td>
</tr>
<tr>
<td>concept_name (a_lang: String): String</td>
<td></td>
</tr>
<tr>
<td>physical_paths: Set&lt;String&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>CLASS</strong></td>
<td><strong>ARCHETYPE</strong></td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>logical_paths</strong> (a_lang: String): Set&lt;String&gt;</td>
<td>Set of language-dependent paths extracted from archetype. Paths obey the same syntax as physical_paths, but with node_ids replaced by their meanings from the ontology.</td>
</tr>
<tr>
<td><strong>is_specialised</strong>: Boolean</td>
<td>True if this archetype is a specialisation of another.</td>
</tr>
<tr>
<td><strong>specialisation_depth</strong>: Integer</td>
<td>Specialisation depth of this archetype; larger than 0 if this archetype has a parent. Derived from ontology.specialisation_depth.</td>
</tr>
<tr>
<td><strong>node_ids_valid</strong>: Boolean</td>
<td>True if every node_id found on a C_OBJECT node is found in ontology.term_codes.</td>
</tr>
<tr>
<td><strong>internal_references_valid</strong>: Boolean</td>
<td>True if every ARCHETYPE_INTERNAL_REF target_path refers to a legitimate node in the archetype definition.</td>
</tr>
<tr>
<td><strong>constraint_references_valid</strong>: Boolean</td>
<td>True if every CONSTRAINT_REF reference found on a C_OBJECT node in the archetype definition is found in ontology.constraint_codes.</td>
</tr>
<tr>
<td><strong>is_valid</strong>: Boolean</td>
<td>True if the archetype is valid overall; various tests should be used, including checks on node_ids, internal references, and constraint references.</td>
</tr>
</tbody>
</table>

**Invariant**

- archetype_id_validity: archetype_id /= Void
- uid_validity: uid /= Void implies not uid.is_empty
- version_validity: version /= Void and then version.is_equal(archetype_id.version_id)
- original_language_valid: original_language /= void and then language /= Void and then code_set(“languages”).has(original_language)
- description_exists: description /= Void
- definition_exists: definition /= Void
- ontology_exists: ontology /= Void
- Specialisation_validity: is_specialised implies specialisation_depth > 0
### 2.4.3 VALIDITY_KIND Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>VALIDITY_KIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>An enumeration of three values which may commonly occur in constraint models.</td>
</tr>
<tr>
<td>Use</td>
<td>Use as the type of any attribute within this model, which expresses constraint on some attribute in a class in a reference model. For example to indicate validity of Date/Time fields.</td>
</tr>
</tbody>
</table>

#### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const mandatory: Integer = 1001</code></td>
<td>Constant to indicate mandatory presence of something</td>
</tr>
<tr>
<td><code>const optional: Integer = 1002</code></td>
<td>Constant to indicate optional presence of something</td>
</tr>
<tr>
<td><code>const disallowed: Integer = 1003</code></td>
<td>Constant to indicate disallowed presence of something</td>
</tr>
<tr>
<td><code>value: Integer</code></td>
<td>Actual value</td>
</tr>
</tbody>
</table>

#### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>valid_validity (a_validity: Integer): Boolean</code></td>
<td>Function to test validity values.</td>
</tr>
</tbody>
</table>

#### Invariant

| Validity: valid_validity(value) |

---

© 2004-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
2.5 Constraint Model Package

2.5.1 Overview

FIGURE 5 illustrates the class model of an archetype definition. This model is completely generic, and is designed to express the semantics of constraints on instances of classes which are themselves described in UML (or a similar object-oriented meta-model). Accordingly, the major abstractions in this model correspond to major abstractions in object-oriented formalisms, including several variations of the notion of ‘object’ and the notion of ‘attribute’. The notion of ‘object’ rather than ‘class’ or ‘type’ is used because archetypes are about constraints on data (i.e. ‘instances’, or ‘objects’) rather than models, which are constructed from ‘classes’.

One way to comprehend the model is via the following statements that can be made about it.

- Any archetype definition is an instance of a C_COMPLEX_OBJECT, which can be thought of as expressing constraints on an object that is of some particular type (recorded in the attribute rm_type_name) in a reference model, and which is larger than a simple instance of a primitive type such as String or Integer.
- A C_COMPLEX_OBJECT consists of attributes of type C_ATTRIBUTE, which are constraints on the attributes (i.e. any property, including relationships) of the reference model type. Accordingly, each C_ATTRIBUTE records the name of the constrained attribute (in rm_attr_name), the existence and cardinality expressed by the constraint (depending on whether the attribute it constrains is a multiple or single relationship), and the constraint on the object to which this C_ATTRIBUTE refers via its children attribute (according to its reference model) in the form of further C_OBJECTs.
- The key subtypes of C_OBJECT, are C_COMPLEX_OBJECT (described above) C_PRIMITIVE_OBJECT (constraints on instances of primitive types such as String, Integer, Boolean and Date).
- The other subtypes of C_OBJECT, namely, ARCHETYPE_SLOT, ARCHETYPE_INTERNAL_REF and CONSTRAINT_REF are used to express, respectively, a ‘slot’ where further archetypes can be used to continue describing constraints; a reference to a part of the current archetype that expresses exactly the same constraints needed at another point; and a reference to a constraint on a constraint defined in the archetype ontology, which in turn points to an external knowledge resource, such as a terminology.
- All nodes in an archetype constraint structure are instances of the supertype ARCHETYPE_CONSTRAINT, which provides a number of important common features to all nodes.

2.5.2 Semantics

The effect of the model is to create archetype description structures that are a hierarchical alternation of object and attribute constraints, as shown in FIGURE 3. This structure can be seen by inspecting an ADL archetype, or by viewing an archetype in the openEHR ADL workbench [9], and is a direct consequence of the object-oriented principle that classes consist of properties, which in turn have types that are classes. (To be completely correct, types do not always correspond to classes in an object model, but it does not make any difference here). The repeated object/attribute hierarchical structure of an archetype provides the basis for using paths to reference any node in an archetype. Archetype paths follow a syntax that is a subset of the W3C Xpath syntax.
FIGURE 5 openehr.am.archetype.constraint_model Package
All Node Types
A small number of properties is defined for all node types. The any_allowed flag set on a node indicates that any value permitted by the reference model for the attribute or type in question is allowed by the archetype; its use permits the logical idea of a completely “open” constraint to be simply expressed, avoiding the need for any further substructure. The path feature computes the path to the current node from the root of the archetype, while the has_path function indicates whether a given path can be found in an archetype. The is_valid function indicates whether the current node and all subnodes are internally valid according to the semantics of this archetype model.

Attribute Nodes
Constraints on attributes are represented by instances of the two subtypes of \texttt{C\_ATTRIBUTE}: \texttt{C\_SINGLE\_ATTRIBUTE} and \texttt{C\_MULTIPLE\_ATTRIBUTE}. For both subtypes, the common constraint is whether the corresponding instance (defined by the \texttt{rm\_attribute\_name} attribute) must exist. Both subtypes have a list of children, representing constraints on the object value(s) of the attribute.

Single-valued attributes (such as Person.date_of_birth: Date) are constrained by instances of the type \texttt{C\_SINGLE\_ATTRIBUTE}, which uses the children to represent multiple alternative object constraints for the attribute value.

Multiply-valued attributes (such as Person.contacts: List<Contact>) are constrained by an instance of \texttt{C\_MULTIPLE\_ATTRIBUTE}, which allows multiple co-existing member objects of the container value of the attribute to be constrained, along with a cardinality constraint, describing ordering and uniqueness of the container. FIGURE 6 illustrates the two possibilities.

<table>
<thead>
<tr>
<th>Single-valued C_ATTRIBUTE with alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple-valued C_ATTRIBUTE with collection (LIST.items)</td>
</tr>
</tbody>
</table>

FIGURE 6 Single and Multiple-valued C\_ATTRIBUTES

The need for both existence and cardinality constraints in the \texttt{C\_MULTIPLE\_ATTRIBUTE} class deserves some explanation, especially as the meanings of these notions are often confused in object-oriented literature. Quite simply, an existence constraint indicates whether an object will be found in a given attribute field, while a cardinality constraint indicates what the valid membership of a container object is. Cardinality is only required for container objects such as List<T>, Set<T> and so on, whereas existence is always required. If both are used, the meaning is as follows: the existence constraint says whether the container object will be there (at all), while the cardinality constraint says how many items must be in the container, and whether it acts logically as a list, set or bag.
Primitive Types
Constraints on primitive types are defined by the classes inheriting from `C_PRIMITIVE`, namely `C_STRING`, `C_INTEGER` and so on. These types do not inherit from `ARCHETYPE_CONSTRAINT`, but rather are related by association, in order to allow them to have the simplest possible definitions, independent even from the rest of ADL, in the hope of acceptance in heath standardisation organisations. Technically, avoiding inheritance from `ARCHETYPE_CONSTRAINT / C_PRIMITIVE_OBJECT` into these base types (in other words, coalescing the classes `C_PRIMITIVE_OBJECT` and `C_PRIMITIVE`) does not pose a problem, but could be effected at a later date if desired.

Constraint References
A `CONSTRAINT_REF` is really a proxy for a set of constraints on an object that would normally occur at a particular point in the archetype as a `C_COMPLEX_OBJECT`, but where the actual definition of the constraints is outside the archetype definition proper, and is instead expressed in the binding of the constraint reference (e.g. ‘ac0004’) to a query or expression into an external service (such as an ontology or terminology service). The result of the query could be something like:

- a set of allowed `CODED_TERMS` e.g. the types of hepatitis
- an `INTERVAL<QUANTITY>` forming a reference range
- a set of units or properties or other numerical item

**To Be Determined:** whether this approach should be used instead:

**To Be Determined:** The other problem is that the `CONSTRAINT_REF` could probably stand for a `C_PRIMITIVE_OBJECT`, such as a plain `C_string` or `C_integer` (which can still be a little bit complex - e.g. a `Interval<Integer>`).

**To Be Determined:** Following on logically from this, a more correct modelling possibility might be to introduce a common parent for `C_COMPLEX_OBJECT` and `C_PRIMITIVE_OBJECT` which corresponds to the idea of `C_OBJECTs 'defined by value in the archetype'` (as opposed to defined elsewhere, like in binding query to a terminology, or else in an entirely different archetype, which is what the slot gives you). If I introduced such a type, then `CONSTRAINT_REF` should have a property called something like 'proxy_for' or 'equivalent', which points to this new type, allowing it to stand for either a primitive or complex constraint structure. Now that you have driven me to think of that, I see it as being quite a good improvement - maybe Andrew will have feedback on it.

**To Be Determined:** Another parent is probably needed for `C_DOMAIN_TYPE` and `C_PRIMITIVE`, to coalesce their leaf attributes assumed_value etc. This would probably require merging `C_PRIMITIVE_OBJECT` and `C_PRIMITIVE`.

Assertions
The `C_ATTRIBUTE` and subtypes of `C_OBJECT` enable constraints to be expressed in a structural fashion. In addition to this, any instance of a `C_COMPLEX_OBJECT` may include one or more invariants. Invariants are statements in a form of predicate logic, which can be used to state constraints on parts of an object. They are not needed to state constraints on a single attribute (since this can be done with an appropriate `C_ATTRIBUTE`), but are necessary to state constraints on more than one attribute, such as a constraint that ‘systolic pressure should be >= diastolic pressure’ in a blood pressure measurement archetype. Invariants are expressed using a syntax derived from the OMG’s OCL syntax (adapted for use with objects rather than classes).

**To Be Continued:** give decent example
Assertions are also used in ARCHETYPE_SLOTS, in order to express the ‘included’ and ‘excluded’ archetypes for the slot. In this case, each assertion is an expression that refers to parts of other archetypes, such as its identifier (e.g. ‘include archetypes with short_concept_name matching xxxx’). Assertions are modelled here as a generic expression tree of unary prefix and binary infix operators. Examples of archetype slots in ADL syntax are given in the openEHR ADL document.

**Node_id and Paths**

The node_id attribute in the class C_OBJECT, inherited to all subtypes, is of great importance in the archetype constraint model. It has two functions:

- it allows archetype object constraint nodes to be individually identified, and in particular, guarantees sibling node unique identification;
- it is the main link between the archetype definition (i.e. the constraints) and the archetype ontology, because each node_id is a ‘term code’ in the ontology.

The existence of node_ids in an archetype is what allows archetype paths to be created, which refer to each node. Not every node in the archetype needs a node_id, if it does not need to be addressed using a path; any leaf or near-leaf node which has no sibling nodes from the same attribute can safely have no node_id.

**Domain-specific Extensions**

The main part of the archetype constraint model allows any type in a reference model to be archetyped - i.e. constrained - in a standard way, which is to say, by a regular cascade of C_COMPLEX_OBJECT / C_ATTRIBUTE / C_PRIMITIVE_OBJECT objects. This generally works well, especially for ‘outer’ container types in models. However, it occurs reasonably often that lower level logical ‘leaf’ types need special constraint semantics that are not conveniently achieved with the standard approach. To enable such classes to be integrated into the generic constraint model, the class C_DOMAIN_TYPE is included. This enables the creation of specific “C_” classes, inheriting from C_DOMAIN_TYPE, which represent custom semantics for particular reference model types. For example, a class called C_QUANTITY might be created which has different constraint semantics from the default effect of a C_COMPLEX_OBJECT / C_ATTRIBUTE cascade representing such constraints in the generic way (i.e. systematically based on the reference model). An example of domain-specific extension classes is shown in Domain-specific Extension Example on page 47.

**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 contain an optional protocol section 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 value for patient position. Amongst clinicians, basic assumptions are nearly always made for such things: in general practice, the position could always safely be assumed to be “sitting” if not otherwise stated; in the hospital setting, “lying” would be the normal assumption. The assumed values feature of archetypes allows such assumptions 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 definable at the leaf level only, which appears to be adequate for all purposes described to date; accordingly, they appear in descendants of C_PRIMITIVE and also C_DOMAIN_TYPE.

The notion of assumed values is distinct from that of ‘default values’. The latter is a local requirement, and as such is stated in templates; default values do appear in data, while assumed values don’t.
2.5.3 ARCHETYPE_CONSTRAINT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE_CONSTRAINT (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Archetype equivalent to LOCATABLE class in openEHR Common reference model. Defines common constraints for any inheritor of LOCATABLE in any reference model.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>any_allowed</td>
<td>Boolean</td>
<td>True if any instance value of this type is considered valid in this archetype. Allows completely ‘open’ constraints to be expressed without requiring any further structure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_valid</td>
<td>Boolean</td>
<td>True if this node (and all its sub-nodes) is a valid archetype node for its type. This function should be implemented by each subtype to perform semantic validation of itself, and then call the is_valid function in any sub-parts, and generate the result appropriately.</td>
</tr>
<tr>
<td>path</td>
<td>String</td>
<td>Path of this node relative to root of archetype.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>has_path</th>
<th>(a_path: String): Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>require</td>
<td>a_path /= Void</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>is_subset_of</th>
<th>(other: ARCHETYPE_CONSTRAINT): Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>require</td>
<td>other /= Void</td>
</tr>
</tbody>
</table>

| Invariant | path_exists: path /= Void |

To Be Continued:  Note: is_subset_of is relatively easy to evaluate for structures, but acNNNN constraints and assertions will be harder, and will most likely require evaluation in a subsumptive environment like OWL.

2.5.4 C_ATTRIBUTE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_ATTRIBUTE(abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract model of constraint on any kind of attribute node.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
</table>

Date of Issue: 10 Nov 2005  Page 22 of 53  Editor:T Beale

© 2004-2006 The openEHR Foundation  email: info@openEHR.org  web: http://www.openEHR.org
### 2.5.5 C_SINGLE_ATTRIBUTE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_ATTRIBUTE(abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm_attribute_name: String</td>
<td>Reference model attribute within the enclosing type represented by a C_OBJECT.</td>
</tr>
<tr>
<td>existence: Interval&lt;Integer&gt;</td>
<td>Constraint on every attribute, regardless of whether it is singular or of a container type, which indicates whether its target object exists or not (i.e. is mandatory or not).</td>
</tr>
<tr>
<td>children: List&lt;C_OBJECT&gt;</td>
<td>Child C_OBJECT nodes. Each such node represents a constraint on the type of this attribute in its reference model. Multiples occur both for multiple items in the case of container attributes, and alternatives in the case of singular attributes.</td>
</tr>
</tbody>
</table>

**Invariant**

- **Rm_attribute_name_valid**: rm_attribute_name /= Void and then not rm_attribute_name.is_empty
- **Existence_set**: existence /= Void and then (existence.lower >= 0 and existence.upper <= 1)
- **Children_validity**: any_allowed xor children /= Void

### 2.5.6 C_MULTIPLE_ATTRIBUTE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_MULTIPLE_ATTRIBUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Concrete model of constraint on any kind of attribute node.</td>
</tr>
</tbody>
</table>

**Attributes**

- **cardinality**: CARDINALITY
  - Cardinality of this attribute constraint, if it constrains a container attribute.

**Functions**
## 2.5.7 CARDINALITY Class

### Purpose
Express constraints on the cardinality of container objects which are the values of multiply-valued attributes, including uniqueness and ordering, providing the means to state that a container acts like a logical list, set or bag. The cardinality cannot contradict the cardinality of the corresponding attribute within the relevant reference model.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_ordered</td>
<td>Boolean</td>
<td>True if the members of the container attribute to which this cardinality refers are ordered.</td>
</tr>
<tr>
<td>is_unique</td>
<td>Boolean</td>
<td>True if the members of the container attribute to which this cardinality refers are unique.</td>
</tr>
<tr>
<td>interval</td>
<td>Interval&lt;Integer&gt;</td>
<td>The interval of this cardinality.</td>
</tr>
</tbody>
</table>

#### Invariant

- **Validity:** not interval.lower_unbounded

---

© 2004-2006 The openEHR Foundation

e-mail: info@openEHR.org web: http://www.openEHR.org

Date of Issue: 10 Nov 2005

Page 24 of 53

Editor: T Beale
# 2.5.8 C_OBJECT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th><strong>C_OBJECT (abstract)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Abstract model of constraint on any kind of object node.</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><code>rm_type_name</code>: String</td>
<td>Reference model type that this node corresponds to.</td>
</tr>
<tr>
<td><code>occurrences</code>: Interval&lt;Integer&gt;</td>
<td>Occurrences of this object node in the data, under the owning attribute. Upper limit can only be greater than 1 if owning attribute has a cardinality of more than 1.</td>
</tr>
<tr>
<td><code>node_id</code>: String</td>
<td>Semantic id of this node, used to differentiate sibling nodes of the same type. [Previously called ‘meaning’]. Each node_id must be defined in the archetype ontology as a term code.</td>
</tr>
<tr>
<td><code>parent</code>: C_ATTRIBUTE</td>
<td>C_ATTRIBUTE that owns this C_OBJECT.</td>
</tr>
</tbody>
</table>

### Invariant

- `rm_type_name_valid`: rm_type_name /= Void and then not rm_type_name.is_empty
- `node_id_valid`: node_id /= Void and then not node_id.is_empty

# 2.5.9 C_COMPLEX_OBJECT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th><strong>C_COMPLEX_OBJECT</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Constraint on complex objects, i.e. any object that consists of other object constraints.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_OBJECT</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td><code>attributes</code>: Set&lt;C_ATTRIBUTE&gt;</td>
<td>List of constraints on attributes of the reference model type represented by this object.</td>
</tr>
<tr>
<td><code>invariants</code>: Set&lt;ASSERTION&gt;</td>
<td>Invariant statements about this object. Statements are expressed in first order predicate logic, and usually refer to at least two attributes.</td>
</tr>
</tbody>
</table>

### Invariant

- `attributes_valid`: any_allowed xor (attributes /= Void and not attributes.is_empty)
- `invariant_consistency`: any_allowed implies invariants = Void
- `invariants_valid`: invariants /= Void implies not invariants.is_empty
2.5.10 ARCHETYPE_SLOT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE_SLOT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Constraint describing a ‘slot’ where another archetype can occur.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_OBJECT</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td></td>
<td><strong>includes</strong>: Set &lt;ASSERTION&gt;</td>
</tr>
<tr>
<td></td>
<td><strong>excludes</strong>: Set&lt;ASSERTION&gt;</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td><strong>includes_valid</strong>: includes /= Void implies not includes.is_empty</td>
</tr>
<tr>
<td></td>
<td><strong>excludes_valid</strong>: excludes /= Void implies not excludes.is_empty</td>
</tr>
<tr>
<td></td>
<td><strong>validity</strong>: any_allowed xor includes /= Void or excludes /= Void</td>
</tr>
</tbody>
</table>

2.5.11 ARCHETYPE_INTERNAL_REF Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE_INTERNAL_REF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>A constraint defined by proxy, using a reference to an object constraint defined elsewhere in the same archetype.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_OBJECT</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td></td>
<td><strong>target_path</strong>: String</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td><strong>Consistency</strong>: not any_allowed</td>
</tr>
<tr>
<td></td>
<td><strong>target_path_valid</strong>: target_path /= Void and then not target_path.is_empty</td>
</tr>
<tr>
<td></td>
<td>-- and then ultimate_root.has_path(target_path)</td>
</tr>
</tbody>
</table>

2.5.12 CONSTRAINT_REF Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>CONSTRAINT_REF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Reference to a constraint described in the same archetype, but outside the main constraint structure. This is used to refer to constraints expressed in terms of external resources, such as constraints on terminology value sets.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_OBJECT</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 2.5.13 **C_PRIMITIVE_OBJECT Class**

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_PRIMITIVE_OBJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constraint on a primitive type.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_OBJECT</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>item:</td>
<td>C_PRIMITIVE</td>
</tr>
<tr>
<td>Invariant</td>
<td>item_exists: any_allowed xor item /= Void</td>
</tr>
</tbody>
</table>
## 2.5.14 C_DOMAIN_TYPE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DOMAIN_TYPE (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent type of domain-specific constrainer types, to be defined in external packages.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_OBJECT</td>
</tr>
</tbody>
</table>

### Abstract

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_value: ANY</td>
<td>Generate a default value from this constraint object</td>
</tr>
<tr>
<td>has_assumed_value: Boolean</td>
<td>True if there is an assumed value</td>
</tr>
<tr>
<td>assumed_value: like</td>
<td>Value to be assumed if none sent in data</td>
</tr>
<tr>
<td>default_value</td>
<td></td>
</tr>
<tr>
<td>valid_value (a_value: like default_value): Boolean require a_value /= Void</td>
<td>True if a_value is valid with respect to constraint expressed in concrete instance of this type.</td>
</tr>
<tr>
<td>standard_representation:</td>
<td>Standard form of constraint</td>
</tr>
<tr>
<td>C_COMPLEX_OBJECT</td>
<td></td>
</tr>
</tbody>
</table>

### Invariant

`Assumed_value_valid: valid_value(assumed_value)`
2.6 The Assertion Package

2.6.1 Overview

Assertions are expressed in archetypes in typed first-order predicate logic (FOL). They are used in two places: to express archetype slot constraints, and to express invariants in complex object constraints. In both of these places, their role is to constrain something inside the archetype. Constraints on external resources such as terminologies are expressed in the constraint binding part of the archetype ontology, described in section 2.8 on page 43.

2.6.2 Semantics

The concrete syntax of assertion statements in archetypes is designed to be compatible with the OMG Object Constraint Language (OCL) [10]. Archetype assertions are essentially statements which contain the following elements:

- **variables**, which are attribute names, or ADL paths terminating in attribute names (i.e. equivalent of referencing class feature in a programming language);
- **manifest constants** of any primitive type, plus date/time types
- **arithmetic operators**: +, *, -, /, ^ (exponent), % (modulo division)
- **relational operators**: >, <, >=, <=, =, !=, matches
- **boolean operators**: not, and, or, xor
- **quantifiers** applied to container variables: for_all, exists

The written syntax of assertions is defined in the openEHR ADL document. The package described here is currently designed to allow the representation of a general-purpose binary expression tree, as would be generated by a parser. This may be replaced in the future by a more specific model, if needed. The assertion package is illustrated below in FIGURE 7.

![FIGURE 7 The openehr.am.archetype.assertion package](attachment:image.png)
This relatively simple model of expressions is sufficiently powerful for representing FOL expressions on archetype structures, although it could clearly be more heavily subtyped.

### 2.6.3 ASSERTION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ASSERTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Structural model of a typed first order predicate logic assertion, in the form of an expression tree, including optional variable definitions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag: String</td>
<td></td>
<td>Expression tag, used for differentiating multiple assertions.</td>
</tr>
<tr>
<td>expression: EXPR_ITEM</td>
<td></td>
<td>Root of expression tree.</td>
</tr>
<tr>
<td>string_expression: String</td>
<td></td>
<td>String form of expression, in case an expression evaluator taking String expressions is used for evaluation.</td>
</tr>
<tr>
<td>variables: List&lt;ASSERTION_VARIABLE&gt;</td>
<td></td>
<td>Definitions of variables used in the assertion expression.</td>
</tr>
</tbody>
</table>

**Invariant**

- Tag_valid: tag /= Void implies not tag.is_empty
- Expression_valid: expression /= Void and then expression.type.is_equal(“BOOLEAN”)

### 2.6.4 EXPR_ITEM Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXPR_ITEM (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Abstract parent of all expression tree items.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: String</td>
<td></td>
<td>Type name of this item. For leaf nodes, must be the name of a primitive type, or else a reference model type. The type for any relational or boolean operator will be “BOOLEAN”, while the type for any arithmetic operator, will be “REAL” or “INTEGER”</td>
</tr>
</tbody>
</table>

**Invariant**

- Type_valid: type /= Void and then not type.is_empty
2.6.5 **EXPR_LEAF Class**

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXPR_LEAF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Expression tree leaf item</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>EXPR_ITEM</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>item: ANY</td>
<td>The value referred to; a manifest constant, an attribute path, or a C_PRIMITIVE. [Future: possibly function names as well, even if not constrained in the archetype - as long as they are in the reference model].</td>
</tr>
<tr>
<td>reference_type: String</td>
<td>Type of reference: “constant”, “attribute”, “function”</td>
</tr>
</tbody>
</table>

**Invariant**

Item_valid: item /= Void

2.6.6 **EXPR_OPERATOR Class**

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXPR_OPERATOR (abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Abstract parent of operator types.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>EXPR_ITEM</td>
</tr>
</tbody>
</table>

**Attributes**

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator: OPERATOR_KIND</td>
<td>Code of operator.</td>
</tr>
<tr>
<td>precedence_overridden: Boolean</td>
<td>True if the natural precedence of operators is overridden in the expression represented by this node of the expression tree. If True, parentheses should be introduced around the totality of the syntax expression corresponding to this operator node and its operands.</td>
</tr>
</tbody>
</table>

**Invariant**

2.6.7 **EXPR_UNARY_OPERATOR Class**

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXPR_UNARY_OPERATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Unary operator expression node.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>EXPR_OPERATOR</td>
</tr>
</tbody>
</table>
### 2.6.8 EXPR_BINARY_OPERATOR Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>EXPR_BINARY_OPERATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Binary operator expression node.</td>
</tr>
<tr>
<td>Inherit</td>
<td>EXPR_OPERATOR</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td>left_operand: EXPR_ITEM</td>
</tr>
<tr>
<td></td>
<td>right_operand: EXPR_ITEM</td>
</tr>
<tr>
<td>Invariant</td>
<td>left_operand_valid: operand /= Void</td>
</tr>
<tr>
<td></td>
<td>right_operand_valid: operand /= Void</td>
</tr>
</tbody>
</table>
## 2.6.9 OPERATOR_KIND Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>OPERATOR_KIND</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Enumeration type for operator types in assertion expressions</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Use as the type of operators in the Assertion package, or for related uses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op_eq</strong>: Integer = 2001</td>
<td></td>
<td>Equals operator (‘=’ or ‘==’)</td>
</tr>
<tr>
<td><strong>op_ne</strong>: Integer = 2002</td>
<td></td>
<td>Not equals operator (‘!=’ or ‘=/’ or ‘&lt;&gt;’)</td>
</tr>
<tr>
<td><strong>op_le</strong>: Integer = 2003</td>
<td></td>
<td>Less-than or equals operator (‘&lt;’ or ‘&lt;=’)</td>
</tr>
<tr>
<td><strong>op_lt</strong>: Integer = 2004</td>
<td></td>
<td>Less-than operator (‘&lt;’)</td>
</tr>
<tr>
<td><strong>op_ge</strong>: Integer = 2005</td>
<td></td>
<td>Greater-than or equals operator (‘&gt;’ or ‘&gt;=' or ‘&gt;=’)</td>
</tr>
<tr>
<td><strong>op_gt</strong>: Integer = 2006</td>
<td></td>
<td>Greater-than operator (‘&gt;’)</td>
</tr>
<tr>
<td><strong>op_matches</strong>: Integer = 2007</td>
<td></td>
<td>Matches operator (‘matches’ or ‘is_in’)</td>
</tr>
<tr>
<td><strong>op_not</strong>: Integer = 2010</td>
<td></td>
<td>Not logical operator</td>
</tr>
<tr>
<td><strong>op_and</strong>: Integer = 2011</td>
<td></td>
<td>And logical operator</td>
</tr>
<tr>
<td><strong>op_or</strong>: Integer = 2012</td>
<td></td>
<td>Or logical operator</td>
</tr>
<tr>
<td><strong>op_xor</strong>: Integer = 2013</td>
<td></td>
<td>Xor logical operator</td>
</tr>
<tr>
<td><strong>op_implies</strong>: Integer = 2014</td>
<td></td>
<td>Implies logical operator</td>
</tr>
<tr>
<td><strong>op_for_all</strong>: Integer = 2015</td>
<td></td>
<td>For-all quantifier operator</td>
</tr>
<tr>
<td><strong>op_exists</strong>: Integer = 2016</td>
<td></td>
<td>Exists quantifier operator</td>
</tr>
<tr>
<td><strong>op_plus</strong>: Integer = 2020</td>
<td></td>
<td>Plus operator (‘+’)</td>
</tr>
<tr>
<td><strong>op_minus</strong>: Integer = 2021</td>
<td></td>
<td>Minus operator (‘-’)</td>
</tr>
<tr>
<td><strong>op_multiply</strong>: Integer = 2022</td>
<td></td>
<td>Multiply operator (‘*’)</td>
</tr>
<tr>
<td><strong>op_divide</strong>: Integer = 2023</td>
<td></td>
<td>Divide operator (‘/’)</td>
</tr>
<tr>
<td>CLASS</td>
<td>OPERATOR_KIND</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>op_exp: Integer = 2024</td>
<td>Exponent operator (‘^’)</td>
<td></td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>value: Integer</td>
<td>Actual value of this instance</td>
</tr>
</tbody>
</table>

### Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid_operator (an_op: Integer): Boolean</td>
<td>Function to test operator values.</td>
</tr>
<tr>
<td>ensure an_op &gt;= op_eq and an_op &lt;= op_exp</td>
<td></td>
</tr>
</tbody>
</table>

### Invariant

**Validity:** valid_operator(value)
2.7 The Primitive Package

Ultimately any archetype definition will devolve down to leaf node constraints on instances of primitive types. The primitive package, illustrated in FIGURE 8, defines the semantics of constraint on such types. Most of the types provide at least two alternative ways to represent the constraint; for example the \texttt{C\_DATE} type allows the constraint to be expressed in the form of a pattern (defined in the ADL specification) or an \texttt{Interval<Date>}. Note that the interval form of dates is probably only useful for historical date checking (e.g. the date of an antique or a particular batch of vaccine), rather than constraints on future date/times.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig8.png}
\caption{The openehr.am.archetype.primitive Package}
\end{figure}
### 2.7.1 C_PRIMITIVE Class

**Purpose**
Constraint on instances of Boolean.

**Use**
Both attributes cannot be set to False, since this would mean that the Boolean value being constrained cannot be True or False.

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_value</td>
<td>ANY</td>
<td>Generate a default value from this constraint object</td>
</tr>
<tr>
<td>has_assumed_value</td>
<td>Boolean</td>
<td>True if there is an assumed value</td>
</tr>
<tr>
<td>assumed_value</td>
<td>like</td>
<td>Value to be assumed if none sent in data</td>
</tr>
<tr>
<td>valid_value</td>
<td>(a_value: like</td>
<td>True if a_value is valid with respect to constraint expressed in concrete instance of this type.</td>
</tr>
<tr>
<td></td>
<td>default_value)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>require</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_value /= Void</td>
<td></td>
</tr>
<tr>
<td>standard_representation</td>
<td>C_COMPLEX_OBJECT</td>
<td>Standard form of constraint</td>
</tr>
</tbody>
</table>

**Invariant**

\[
Assumed\_value\_valid: valid\_value(assumed\_value)
\]

### 2.7.2 C_BOOLEAN Class

**Purpose**
Constraint on instances of Boolean.

**Use**
Both attributes cannot be set to False, since this would mean that the Boolean value being constrained cannot be True or False.

**Inherit**
C_PRIMITIVE

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>true_valid</td>
<td>Boolean</td>
<td>True if the value True is allowed</td>
</tr>
<tr>
<td>false_valid</td>
<td>Boolean</td>
<td>True if the value False is allowed</td>
</tr>
<tr>
<td>assumed_value</td>
<td>Boolean</td>
<td>The value to assume if this item is not included in data, due to being part of an optional structure.</td>
</tr>
</tbody>
</table>
2.7.3 C_STRING Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constraint on instances of STRING</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_PRIMITIVE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern: String</td>
<td>Regular expression pattern for proposed instances of String to match.</td>
<td></td>
</tr>
<tr>
<td>list: Set&lt;String&gt;</td>
<td>Set of Strings specifying constraint</td>
<td></td>
</tr>
<tr>
<td>list_open: Boolean</td>
<td>True if the list is being used to specify the constraint but is not considered exhaustive.</td>
<td></td>
</tr>
<tr>
<td>assumed_value: String</td>
<td>The value to assume if this item is not included in data, due to being part of an optional structure.</td>
<td></td>
</tr>
</tbody>
</table>

Invariant

Consistency: pattern /= Void xor list /= Void
pattern_exists: pattern /= Void implies not pattern.is_empty

To Be Continued: TB: is list_open really useful? If the list is open, then what's the difference from 'any_allowed'?

2.7.4 C_INTEGER Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constraint on instances of Integer.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_PRIMITIVE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>list: Set&lt;Integer&gt;</td>
<td>Set of Integers specifying constraint</td>
<td></td>
</tr>
<tr>
<td>range: Interval&lt;Integer&gt;</td>
<td>Range of Integers specifying constraint</td>
<td></td>
</tr>
<tr>
<td>assumed_value: Integer</td>
<td>The value to assume if this item is not included in data, due to being part of an optional structure.</td>
<td></td>
</tr>
</tbody>
</table>
### 2.7.5 C_REAL Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_REAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Constraint on instances of Real.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_PRIMITIVE</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>list</td>
<td>Set&lt;Real&gt;</td>
</tr>
<tr>
<td>range</td>
<td>Interval&lt;Real&gt;</td>
</tr>
<tr>
<td>assumed_value</td>
<td>Real</td>
</tr>
<tr>
<td><strong>Invariant</strong></td>
<td><strong>Consistency</strong>: list /= Void xor range /= Void</td>
</tr>
</tbody>
</table>

### 2.7.6 C_DATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
<td>Constraint on instances of Date in the form either of a set of validity values, or an actual date range. There is no validity flag for 'year', since it must always be by definition mandatory in order to have a sensible date at all.</td>
</tr>
<tr>
<td><strong>Use</strong></td>
<td>Date ranges are probably only useful for historical dates.</td>
</tr>
<tr>
<td><strong>Inherit</strong></td>
<td>C_PRIMITIVE</td>
</tr>
<tr>
<td><strong>Attributes</strong></td>
<td><strong>Signature</strong></td>
</tr>
<tr>
<td>month_validity</td>
<td>VALIDITY_KIND</td>
</tr>
<tr>
<td>day_validity</td>
<td>VALIDITY_KIND</td>
</tr>
<tr>
<td>timezone_validity</td>
<td>VALIDITY_KIND</td>
</tr>
<tr>
<td>range</td>
<td>Interval&lt;Date&gt;</td>
</tr>
<tr>
<td>assumed_value</td>
<td>Date</td>
</tr>
</tbody>
</table>
### 2.7.7 C_TIME Class

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>minute_validity:</td>
<td>VALIDITY_KIND</td>
<td>Validity of minute in constrained time.</td>
</tr>
<tr>
<td>second_validity:</td>
<td>VALIDITY_KIND</td>
<td>Validity of second in constrained time.</td>
</tr>
<tr>
<td>millisecond_validity:</td>
<td>VALIDITY_KIND</td>
<td>Validity of millisecond in constrained time.</td>
</tr>
<tr>
<td>timezone_validity:</td>
<td>VALIDITY_KIND</td>
<td>Validity of timezone in constrained date.</td>
</tr>
<tr>
<td>range:</td>
<td>Interval&lt;Time&gt;</td>
<td>Interval of Times specifying constraint</td>
</tr>
<tr>
<td>assumed_value:</td>
<td>Time</td>
<td>The value to assume if this item is not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>included in data, due to being part of an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>optional structure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>validity_is_range:</td>
<td>Boolean</td>
<td>True if validity is in the form of a range;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>useful for developers to check which kind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of constraint has been set.</td>
</tr>
</tbody>
</table>

**Invariant**

- **Month_validity Optional**
  - month_validity = (VALIDITY_KIND).optional implies (day_validity = (VALIDITY_KIND).optional or day_validity = (VALIDITY_KIND).disallowed)
- **Month_validity Disallowed**
  - month_validity = (VALIDITY_KIND).disallowed implies day_validity = (VALIDITY_KIND).disallowed
- **Validity is Range**
  - validity_is_range = (range /= Void)

---

The Archetype Object Model

© 2004-2006 The openEHR Foundation
email: info@openEHR.org web: http://www.openEHR.org
CLASS C_DATE_TIME

Purpose
Constraint on instances of Date_Time. There is no validity flag for 'year', since it must always be by definition mandatory in order to have a sensible date/time at all.

Inherit
C_PRIMITIVE

Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>month_validity:</td>
<td>Validity of month in constrained date.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>day_validity:</td>
<td>Validity of day in constrained date.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>hour_validity:</td>
<td>Validity of hour in constrained time.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>minute_validity:</td>
<td>Validity of minute in constrained time.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>second_validity:</td>
<td>Validity of second in constrained time.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>millisecond_validity:</td>
<td>Validity of millisecond in constrained time.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>timezone_validity:</td>
<td>Validity of timezone in constrained date.</td>
</tr>
<tr>
<td>VALIDITY_KIND</td>
<td></td>
</tr>
<tr>
<td>range:</td>
<td>Range of Date_times specifying constraint</td>
</tr>
<tr>
<td>Interval&lt;Date_Time&gt;</td>
<td></td>
</tr>
<tr>
<td>assumed_value:</td>
<td>The value to assume if this item is not included</td>
</tr>
<tr>
<td>Date_Time</td>
<td>in data, due to being part of an optional structure.</td>
</tr>
</tbody>
</table>

CLASS C_TIME


Minute_validity_disallowed: minute_validity = (VALIDITY_KIND).disallowed implies second_validity = (VALIDITY_KIND).disallowed


Second_validity_disallowed: second_validity = (VALIDITY_KIND).disallowed implies millisecond_validity = (VALIDITY_KIND).disallowed

Validity_is_range: validity_is_range = (range /= Void)
### Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>validity_is_range</strong></td>
<td>Boolean</td>
<td>True if validity is in the form of a range; useful for developers to check which kind of constraint has been set.</td>
</tr>
</tbody>
</table>

### Invariant

<table>
<thead>
<tr>
<th>Invariant</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Month_validity_optional</strong></td>
<td>month_validity = (VALIDITY_KIND).optional implies day_validity = (VALIDITY_KIND).optional or day_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Month_validity_disallowed</strong></td>
<td>month_validity = (VALIDITY_KIND).disallowed implies day_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Day_validity_optional</strong></td>
<td>day_validity = (VALIDITY_KIND).optional implies hour_validity = (VALIDITY_KIND).optional or hour_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Day_validity_disallowed</strong></td>
<td>day_validity = (VALIDITY_KIND).disallowed implies hour_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Hour_validity_optional</strong></td>
<td>hour_validity = (VALIDITY_KIND).optional implies minute_validity = (VALIDITY_KIND).optional or minute_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Hour_validity_disallowed</strong></td>
<td>hour_validity = (VALIDITY_KIND).disallowed implies minute_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Minute_validity_optional</strong></td>
<td>minute_validity = (VALIDITY_KIND).optional implies second_validity = (VALIDITY_KIND).optional or second_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Minute_validity_disallowed</strong></td>
<td>minute_validity = (VALIDITY_KIND).disallowed implies second_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Second_validity_optional</strong></td>
<td>second_validity = (VALIDITY_KIND).optional implies millisecond_validity = (VALIDITY_KIND).optional or millisecond_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Second_validity_disallowed</strong></td>
<td>second_validity = (VALIDITY_KIND).disallowed implies millisecond_validity = (VALIDITY_KIND).disallowed</td>
<td></td>
</tr>
<tr>
<td><strong>Validity_is_range</strong></td>
<td>validity_is_range = (range /= Void)</td>
<td></td>
</tr>
</tbody>
</table>
## 2.7.9 C_DURATION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constraint on instances of Duration.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_PRIMITIVE</td>
</tr>
</tbody>
</table>

### Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>range: Interval&lt;Duration&gt;</code></td>
<td>Range of Durations specifying constraint</td>
<td></td>
</tr>
<tr>
<td><code>assumed_value: Duration</code></td>
<td>The value to assume if this item is not included in data, due to being part of an optional structure.</td>
<td></td>
</tr>
</tbody>
</table>

### Invariant

`Range_valid: range /= Void`
2.8 Ontology Package

2.8.1 Overview

All linguistic and terminological entities in an archetype are represented in the ontology part of an archetype, whose semantics are given in the Ontology package, shown below.

An archetype ontology consists of the following things.

- A list of terms defined local to the archetype. These are identified by ‘atNNNN’ codes, and perform the function of archetype node identifiers from which paths are created. There is one such list for each natural language in the archetype. A term ‘at0001’ defined in English as ‘blood group’ is an example.
- A list of external constraint definitions, identified by ‘acNNNN’ codes, for constraints defined external to the archetype, and referenced using an instance of a CONSTRAINT_REF. There is one such list for each natural language in the archetype. A term ‘ac0001’ corresponding to ‘any term which is-a blood group’, which can be evaluated against some external terminology service.
- Optionally, a set of one or more bindings of term definitions to term codes from external terminologies.
- Optionally, a set of one or more bindings of the external constraint definitions to external resources such as terminologies.

2.8.2 Semantics

Specialisation Depth

Any given archetype occurs at some point in a hierarchy of archetypes related by specialisation, where the depth is indicated by the specialisation_depth attribute. An archetype which is not a specialisation of another has a specialisation_depth of 0. Term and constraint codes introduced in the ontology of specialised archetypes (i.e. which did not exist in the ontology of the parent archetype)
are defined in a strict way, using ‘.’ (period) markers. For example, an archetype of specialisation depth 2 will use term definition codes like the following:

- ‘at0001.0.1’ - a term which specialises the ‘at0001’ term from the top parent. An intervening ‘.0’ is required to show that the new term is at depth 2, not depth 1;
- ‘at0001.1.1’ - a term which specialises the term ‘at0001.1’ from the immediate parent, which itself specialises the term ‘at0001’ from the top parent.

This systematic definition of codes enables software to use the structure of the codes to more quickly and accurately make inferences about term definitions up and down specialisation hierarchies. Constraint codes on the other hand do not follow these rules, and exist in a flat code space instead.

**Term and Constraint Definitions**

Local term and constraint definitions are modelled as instances of the class `ARCHETYPE_TERM`, which is a code associated with a list of name/value pairs. For any term or constraint definition, this list must at least include the name/value pairs for the names “text” and “description”. It might also include such things as “provenance”, which would be used to indicate that a term was sourced from an external terminology. The attribute `term_attribute_names` in `ARCHETYPE_ONTOLOGY` provides a list of attribute names used in term and constraint definitions in the archetype, including “text” and “description”, as well as any others which are used in various places.

### 2.8.3 ARCHETYPE_ONTOLOGY Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>ARCHETYPE_ONTOLOGY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Local ontology of an archetype.</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>terminologies_available: Set&lt;String&gt;</td>
<td>List of terminologies to which term or constraint bindings exist in this terminology.</td>
</tr>
<tr>
<td>specialisation_depth: Integer</td>
<td>Specialisation depth of this archetype. Unspecialised archetypes have depth 0, with each additional level of specialisation adding 1 to the specialisation_depth.</td>
</tr>
<tr>
<td>term_codes: List&lt;String&gt;</td>
<td>List of all term codes in the ontology. Most of these correspond to “at” codes in an ADL archetype, which are the node_ids on C_OBJECT descendants. There may be an extra one, if a different term is used as the overall archetype concept_code from that used as the node_id of the outermost C_OBJECT in the definition part.</td>
</tr>
</tbody>
</table>
### constraint_codes:

| List<String> | List of all term codes in the ontology. These correspond to the “ac” codes in an ADL archetype, or equivalently, the CONSTRAINT_REF reference values in the archetype definition. |

### term_attribute_names:

| List<String> | List of ‘attribute’ names in ontology terms, typically includes ‘text’, ‘description’, ‘provenance’ etc. |

### parent_archetype:

| ARCHETYPE | Archetype which owns this ontology. |

## Functions

### has_language(a_lang: String):

| Boolean | True if language ‘a_lang’ is present in archetype ontology. |

### has_terminology(a_terminology_id: String):

| Boolean | True if terminology ‘a_terminology’ is present in archetype ontology. |

### term_definition(a_lang, a_code: String):

| ARCHETYPE_TERM | Term definition for a code, in a specified language. |

### constraint_definition(a_lang, a_code: String):

| ARCHETYPE_TERM | Constraint definition for a code, in a specified language. |

### term_binding(a_terminology_id, a_code: String):

| CODE_PHRASE | Binding of term corresponding to a_code in target external terminology a_terminology_id as a CODE_PHRASE. |
constraint_binding
(a_terminology_id, a_code:
String): String
require
has_terminology(a_terminology_id)
constraint_codes.has(a_code)

Binding of constraint corresponding to
a_code in target external terminology
a_terminology_id, as a string, which is usu-
ally a formal query expression.

terminologies_available_exists: terminologies_available /= void
term_codes_exists: term_codes /= void
constraint_codes_exists: constraint_codes /= void
term_bindings_exists: term_bindings /= void
constraint_bindings_exists: constraint_bindings /= void
term_attribute_names_valid: term_attribute_names /= void and then
term_attribute_names.has(“text”) and term_attribute_names.has(“description”)
concept_code_valid: term_codes.has(concept_code)
Parent_archetype_valid: parent_archetype /= Void and then
parent_archetype.description = Current

2.8.4 ARCHETYPE_TERM Class

Purpose
Representation of any coded entity (term or constraint) in the archetype ontology.

Attributes

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>code: String</td>
<td>Code of this term.</td>
</tr>
<tr>
<td>items: Hash &lt;String, String&gt;</td>
<td>Hash of keys (“text”, “description” etc) and corresponding values.</td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>keys: Set&lt;String&gt;</td>
<td>List of all keys used in this term.</td>
</tr>
</tbody>
</table>

Invariant

| code_valid: code /= void and then not code.is_empty |

CLASS ARCHETYPE_TERM

CLASS ARCHETYPE_ONTOLOGY

CLASS ARCHETYPE_ONTOLOGY

CLASS ARCHETYPE_ONTOLOGY
A Domain-specific Extension Example

A.1 Overview

Domain-specific classes can be added to the archetype constraint model by inheriting from the class `C_DOMAIN_TYPE`. This section provides an example of how domain-specific constraint classes are added to the archetype model.

A.2 Scientific/Clinical Computing Types

FIGURE 10 shows the general approach, used to add constraint classes for commonly used concepts in scientific and clinical computing, such as ‘ordinal’ (used heavily in medicine, particularly in pathology testing), ‘coded term’ (also heavily used in clinical computing) and ‘quantity’, a general scientific measurement concept. The constraint types shown are `C_ORDINAL`, `C_CODED_TEXT` and `C_QUANTITY` which can optionally be used in archetypes to replace the default constraint semantics represented by the use of instances of `C_OBJECT` / `C_ATTRIBUTE` to constrain ordinals, coded terms and quantities. The following model is intended only as an example, and does not try to define any normative semantics of the particular constraint types shown.

![FIGURE 10 Example Domain-specific Package](chart_url)
B Using Archetypes with Diverse Reference Models

B.1 Overview
The archetype model described in this document can be used with any reference model which is expressed in UML or a similar object-oriented formalism. It can also be used with E/R models. The following section describes is use a number of reference models used in clinical computing.

B.2 Clinical Computing Use
To Be Continued:
  - data types
  - class naming
  - domain archetype semantics versus LCD semantics of exchange models
  - mapping from C_DOMAIN_TYPE subtypes into various RM

B.2.1 openEHR

B.2.2 CEN ENV13606

B.2.3 HL7 Clinical Document Architecture (CDA)

B.2.4 HL7v3 RIM
C References

Publications


Resources


The openEHR Archetype Model

openEHR Archetype Profile

Editors: T Beale

Revision: 0.5.1

Pages: 25

© 2005-2006 The openEHR Foundation

The openEHR foundation

is an independent, non-profit community, facilitating the creation and sharing
of health records by consumers and clinicians via open-source, standards-
based implementations.

Founding Chairman
David Ingram, Professor of Health Informatics, CHIME, University
College London

Founding Members
Dr P Schloeffel, Dr S Heard, Dr D Kalra, D Lloyd, T Beale

Patrons
To Be Announced

email: info@openEHR.org web: http://www.openEHR.org
Copyright Notice

© Copyright openEHR Foundation 2001 - 2006
All Rights Reserved

1. This document is protected by copyright and/or database right throughout the world and is owned by the openEHR Foundation.
2. You may read and print the document for private, non-commercial use.
3. You may use this document (in whole or in part) for the purposes of making presentations and education, so long as such purposes are non-commercial and are designed to comment on, further the goals of, or inform third parties about, openEHR.
4. You must not alter, modify, add to or delete anything from the document you use (except as is permitted in paragraphs 2 and 3 above).
5. You shall, in any use of this document, include an acknowledgement in the form: "© Copyright openEHR Foundation 2001-2006. All rights reserved. www.openEHR.org"
6. This document is being provided as a service to the academic community and on a non-commercial basis. Accordingly, to the fullest extent permitted under applicable law, the openEHR Foundation accepts no liability and offers no warranties in relation to the materials and documentation and their content.
7. If you wish to commercialise, license, sell, distribute, use or otherwise copy the materials and documents on this site other than as provided for in paragraphs 1 to 6 above, you must comply with the terms and conditions of the openEHR Free Commercial Use Licence, or enter into a separate written agreement with openEHR Foundation covering such activities. The terms and conditions of the openEHR Free Commercial Use Licence can be found at http://www.openehr.org/free_commercial_use.htm
**Amendment Record**

<table>
<thead>
<tr>
<th>Issue</th>
<th>Details</th>
<th>Who</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5.1</td>
<td><strong>CR-000200</strong>: Correct Release 1.0 typographical errors. Global changes to this document.</td>
<td>T Beale, D Lloyd</td>
<td>03 Mar 2006</td>
</tr>
<tr>
<td>0.5</td>
<td><strong>CR-000127</strong>: Restructure archetype specifications. Initial Writing.</td>
<td>T Beale</td>
<td>05 Feb 2005</td>
</tr>
</tbody>
</table>

**Acknowledgements**

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

1 Introduction ........................................................................................................... 7
  1.1 Purpose........................................................................................................... 7
  1.2 Related Documents ....................................................................................... 7
  1.3 Status .............................................................................................................. 7
  1.4 Peer review ................................................................................................... 7

2 Overview ............................................................................................................... 9
  2.1 Design Background ....................................................................................... 9
  2.2 Package Structure ......................................................................................... 9

3 Data_types.basic Package .............................................................................. 11
  3.1 Class Descriptions ....................................................................................... 11
  3.1.1 C_DV_STATE Class ............................................................................... 11
  3.1.2 STATE_MACHINE Class ...................................................................... 12
  3.1.3 STATE Class .......................................................................................... 12
  3.1.4 TRANSITION Class .............................................................................. 12

4 Data_types.text Package ............................................................................. 14
  4.1 Overview ....................................................................................................... 14
  4.2 Requirements ............................................................................................... 14
  4.3 Design ........................................................................................................... 15
    4.3.1 Standard ADL Approach ...................................................................... 15
    4.3.2 Terminology-specific Code Constraints .............................................. 16
    4.3.3 Terminology-neutral Code Constraints .............................................. 16
  4.4 Pre-evaluation ............................................................................................... 17
  4.5 Class Descriptions ....................................................................................... 18
    4.5.1 C_DV_CODED_TEXT Class ............................................................... 18

5 Data_types.quantity Package ..................................................................... 19
  5.1 Overview ....................................................................................................... 19
  5.2 Design ........................................................................................................... 19
    5.2.1 Ordinal Type Constraint ...................................................................... 19
    5.2.2 Quantity Type Constraint ................................................................... 20
    5.2.3 Constraining Units .............................................................................. 21
  5.3 Class Definitions ........................................................................................... 22
    5.3.1 C_DV_ORDINAL Class Definition .................................................... 22
    5.3.2 C_DV_QUANTITY Class Definition .................................................. 22
    5.3.3 C_QUANTITY_ITEM Class Definition ............................................ 23
1 Introduction

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

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

1.2 Related Documents
Prerequisite documents for reading this document include:

- The openEHR Archetype Definition Language (ADL)
- The openEHR Archetype Object Model (AOM)

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

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

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

1.4 Peer review
Known omissions or questions are indicated in the text with a “to be determined” paragraph, as follows:

TBD_1: (example To Be Determined paragraph)

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

To Be Continued: more work required

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

2.1 Design Background

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

- The openEHR Reference Model (RM), defining the structure and semantics of information in terms of information models (IMs). The RM models correspond to the ISP RM/ODP information viewpoint, and define the data of openEHR EHR systems. The information model is designed to be invariant in the long term, to minimise the need for software and schema updates.

- The openEHR Archetype Model (AM), defining the structure and semantics of archetypes and templates. The AM consists of the archetype language definition language (ADL), the Archetype Object Model (AOM) and the openEHR Archetype profile (oAP).

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

The purpose of the openEHR Archetype Profile, the subject of this document, is to define custom archetype classes that can be used instead of the AOM generic classes for archetyping certain RM classes.

By way of example, consider the openEHR RM type DV_QUANTITY. The generic AOM enables this to be archetyped with instances of C_COMPLEX_OBJECT and C_ATTRIBUTE. However, this does not always provide the most useful semantics for expressing constraints on DV_QUANTITY. The problem is solved by creating a class C_DV_QUANTITY, a class defining custom constraint semantics for DV_QUANTITY instances, which can be used as an optional replacement for the default C_COMPLEX_OBJECT and related objects. Custom archetype classes can be defined for any type in the reference model. A detailed discussion of this example can be found in the openEHR ADL document.

2.2 Package Structure

The openEHR Archetype Profile model is defined in the package am.openehr_profile, illustrated in FIGURE 1. It is shown in the context of the openEHR am and am.archetype packages. The internal structure of the package mimics the structure of the reference model it profiles, i.e. the openEHR reference model. This is done to make software development easier, even though the package structure may be sparsely populated. Packages need only be defined where there are custom types to be defined; the only ones currently defined are in the data_types package.
FIGURE 1 openehr.am.openehr_profile Package
3 Data_types.basic Package

The am.openehr_profile.basic package, illustrated in FIGURE 2, defines custom types for constraining the RM type DV_STATE.

A example of a state machine to model the state of a medication order is illustrated in FIGURE 3. This state machine is defined by an instance of the class STATE_MACHINE.

3.1 Class Descriptions

3.1.1 C_DV_STATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DV_STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constrainer type for DV_STATE instances. The attribute c_value defines a</td>
</tr>
<tr>
<td></td>
<td>state/event table which constrains the allowed values of the attribute</td>
</tr>
<tr>
<td></td>
<td>value in a DV_STATE instance, as well as the order of transitions between</td>
</tr>
<tr>
<td></td>
<td>values.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_DOMAIN_TYPE</td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td>1..1</td>
<td>value: STATE_MACHINE</td>
</tr>
<tr>
<td>Invariants</td>
<td>value_exists: c_value /= Void</td>
</tr>
</tbody>
</table>
# 3.1.2 STATE_MACHINE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>STATE_MACHINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Definition of a state machine in terms of states, transition events and outputs, and next states.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 states: Set &lt;STATE&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>States_valid: states /= Void and then not states.is_empty</td>
</tr>
</tbody>
</table>

## 3.1.3 STATE Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Definition of one state in a state machine.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 name: String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1..1 transitions: Set &lt;TRANSITION&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name_valid: name /= Void and then not name.is_empty</td>
</tr>
<tr>
<td>Transitions_valid: transitions /= Void and then not transitions.is_empty</td>
</tr>
</tbody>
</table>

## 3.1.4 TRANSITION Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>TRANSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Definition of a state machine transition.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 event: String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..1 guard: String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..1 action: String</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1..1 next_state: STATE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 states: Set &lt;STATE&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>States_valid: states /= Void and then not states.is_empty</td>
</tr>
<tr>
<td>CLASS</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Invariants</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

*Event_valid: event /= Void and then not event.is_empty*

*Action_valid: action /= Void implies not action.is_empty*

*Guard_valid: guard /= Void implies not guard.is_empty*

*Next_state_valid: next_state /= Void*
4 Data_types.text Package

4.1 Overview

The am.openehr_profile.data_types.text package contains custom classes for expressing constraints on instances of the types defined in the rm.data_types.text package. Only one type is currently defined, enabling the constraining of DV_CODED_TEXT instances. It is illustrated in FIGURE 4.

4.2 Requirements

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

Constraints on Names

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

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

Constraints on Values

The second kind of constraint on coded terms is used where terms appear as values. In this case, the intention is to specify a set of allowed terms, for example blood groups, diagnoses which may be relevant in the particular clinical setting, or the characteristics of a lump on palpation. More complex constraints specify that the set of terms is the union of two or more groups (the OR operator in queries), or is a member of a number of groups (the AND operator in queries), or even some more complex combination. In all cases, we can think of the constraint as returning a “candidate set of terms” when evaluated against real terminologies.
A candidate set of terms can be obtained from a terminology in a number of ways. First, via the use of relationships encoded in the terminology, such as: “X is-a-kind-of coronary disease”, where classification relationships such as “is-a-kind-of” are defined in the terminology of interest. Second, by identifying terms which belong in some kind of group or category. Consider a constraint such as “X has-category palpable-body-part” which will return the set of terms which describe palpable body parts. These two methods may be mixed as in “X is-a-kind-of body-part AND has-category palpable”, which uses both a relationship and a category - and is equivalent to the previous category described. Note that a constraint like “X is-a-kind-of body-part” is likely to return a long list of body parts, while the category of “palpable” body-parts would reduce this significantly. Such constraints should only be specified if there is a mechanism to implement the categorisation - this might not be in the terminology but must be available to the terminology service.

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

An example of such a constraint is:

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

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

• a particular term
• a named relationship
• a named category

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

4.3 Design

4.3.1 Standard ADL Approach

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

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

### 4.3.2 Terminology-specific Code Constraints

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

- `terminology_id = <"ICD10">`  
  `code_list = <[F43.1]>` -- post traumatic stress disorder
- `terminology_id = <"ICD10">`  
  `subset = <[xxx]>` -- acute stress reactions
  `code_list = <[F43.00], [F43.01], [F32.02]>` -- acute stress reaction, mild
  `subset = <[xxx]>` -- acute stress reaction, moderate
  `subset = <[xxx]>` -- acute stress reaction, severe
- `terminology_id = <"SNOMED-CT">`  
  `subset = <[xxx]>` -- body structures

### 4.3.3 Terminology-neutral Code Constraints

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

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

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

### Archetype-local Codes

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

```
    code matches {{local::at0016}}
    code matches {{hl7_ClassCode::EVN, OBS}}
    code matches {
        [local::
            at1311, -- Colo-colonic anastomosis
            at1312, -- Ileo-colonic anastomosis
            at1313, -- Colo-anal anastomosis
            at1314, -- Ileo-anal anastomosis
            at1315] -- Colostomy
    }
```

These can all be represented as instances of the class `C_DV_CODED_TEXT` by simply setting `terminology_id` to “local”.

**Abstract Inline Queries**

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

**External bindings in the Archetype Ontology**

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

```
    code matches {{ac0016}} -- type of respiratory illness
    property matches {{ac0034}} -- acceleration
```

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

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

### 4.4 Pre-evaluation

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

#### 4.5.1 C_DV_CODED_TEXT Class

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DV_CODED_TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Express constraints on instances of <code>DV_CODED_TEXT</code>. The attributes <code>terminology_id</code>, <code>code_list</code> and <code>subset</code> are to be used when a particular terminology is targeted. The attribute <code>query</code> is reserved for future possible use, where abstract queries might be possible, which do not mention any terminology. If <code>query</code> is used, the other attributes have no meaning. Only one of the <code>terminology_id</code> and <code>query</code> attributes can be non-void.</td>
</tr>
<tr>
<td>Inherit</td>
<td><code>C_DOMAIN_TYPE</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1 (cond)</td>
<td><code>terminology_id:</code> TERMINOLOGY_ID</td>
<td>Syntax string expressing constraint on allowed primary terms</td>
</tr>
<tr>
<td>0..1 (cond)</td>
<td><code>code_list:</code> List&lt;String&gt;</td>
<td>List of codes; may be empty</td>
</tr>
<tr>
<td>0..1 (cond)</td>
<td><code>subset:</code> String</td>
<td>Optional name of subset in terminology from which codes must come. Only useful for terminologies which support subsetting.</td>
</tr>
<tr>
<td>0..1 (cond)</td>
<td><code>query:</code> String</td>
<td>Constraint in terms of an abstract query expression to be addressed to a terminology server.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invariants</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Terminology_id_valid</code>: <code>terminology_id</code> /= Void implies <code>not</code> (terminology_id.is_empty or code_list = Void)</td>
<td></td>
</tr>
<tr>
<td><code>Subset_valid</code>: <code>subset</code> /= Void implies (terminology_id /= Void and not subset.is_empty)</td>
<td></td>
</tr>
<tr>
<td><code>General_validity</code>: <code>terminology_id</code> /= Void xor <code>query</code> /= Void</td>
<td></td>
</tr>
<tr>
<td><code>Any_allowed_validity</code>: <code>code_list.is_empty</code> implies <code>any_allowed</code></td>
<td></td>
</tr>
</tbody>
</table>
5 Data_types.quantity Package

5.1 Overview

The `am.openehr_profile.data_types.quantity` package is illustrated in FIGURE 5. Two custom types are defined: `C_DV_QUANTITY` and `C_DV_ORDINAL`.

![Diagram](image)

**FIGURE 5** `am.openehr_profile.data_types.quantity` Package

5.2 Design

5.2.1 Ordinal Type Constraint

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

```adl
item matches {
  ORDINAL matches {
    value matches {0}
    symbol matches {
      CODED_TEXT matches {
        code matches {[local::at0014]} -- no heartbeat
      }
    }
  }
  ORDINAL matches {
    value matches {1}
    symbol matches {
      CODED_TEXT matches {
        code matches {[local::at0015]} -- less than 100 bpm
      }
    }
  }
  ORDINAL matches {
    value matches {2}
    symbol matches {
      CODED_TEXT matches {
        code matches {[local::at0016]} -- greater than 100 bpm
      }
    }
  }
}
```
The above says that the allowed values of the attribute value is the set of **ORDINALs** represented by three alternative constraints, each indicating what the numeric value of the ordinal in the series, as well as its symbol, which is a **CODED_TEXT**.

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

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

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

### 5.2.2 Quantity Type Constraint

Another situation in which standard ADL falls short is when the required semantics of constraint are different from those provided by the standard approach. Consider a simple type **QUANTITY**, shown at the top of FIGURE 6, which could be used to represent a person's age in data. A typical ADL constraint to enable **QUANTITY** to be used to represent age in clinical data is shown below, followed by its expression in ADL. The only way to do this in ADL is to use multiple alternatives. While this is a perfectly legal approach, it makes processing by software difficult, since the way such a constraint would be displayed in a GUI would be factored differently.

A more powerful possibility is to introduce a new class into the archetype model, representing the concept “constraint on **QUANTITY**”, which we will call **C_QUANTITY** here. Such a class fits into the class model of archetypes (described in the openEHR Archetype Model document), inheriting from the class **C_DOMAIN_TYPE**. The **C_DV_QUANTITY** class is illustrated in FIGURE 7, and corresponds to

![FIGURE 6 Standard ADL for Constraint on Quantity](image-url)
the way constraints on \texttt{QUANTITY} objects are expressed in user applications, which is to say, a property constraint, and a separate list of units/magnitude pairs.

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 \texttt{QUANTITY} must be an instance of a \texttt{C_QUANTITY}; which can be expressed at the appropriate point in the archetype in the form of a section of dADL - the data syntax used in the archetype ontology.

\begin{verbatim}
value matches {
  C_QUANTITY <
    property = "time"
    list = <
      items = <
        [1] = <
          units = "yr"
          magnitude = |0.0..200.0|
        >
        [2] = <
          units = "mth"
          magnitude = |1.0..36.0|
        >
      >
    >
} 
\end{verbatim}

\textbf{FIGURE 7} Inclusion of a Constraint Object as Data

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

- the dADL section occurs inside the \{\} block where its standard ADL equivalent would have occurred (i.e. no other delimiters or special marks are needed);
- the dADL section must be ‘typed’, i.e. it must start with a type name, which should be a rule-based transform of a reference model type (as described in Adding Type Information on page 28);
- the dADL instance must obey the semantics of the custom type of which it is an instance.

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

\section*{5.2.3 Constraining Units}

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

\begin{verbatim}
definition QUANTITY matches {
  magnitude matches [0.0..500.0]
  units matches [ac0001]
}
ontology
...
items("ac0001") = <query("units", "unit matches ‘FORCE/DISTANCE^2’")>
\end{verbatim}
In the above, the expression “FORCE/DISTANCE^2” is an instance of a code phrase from a terminology called “units”; i.e. most likely a post-coordination from a units term engine.

5.3 Class Definitions

5.3.1 C_DV_ORDINAL Class Definition

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DV_ORDINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Class specifying constraints on instances of DV_ORDINAL. Custom constrainer type for instances of DV_ORDINAL.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_DV_ORDERED</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 list: Set&lt;DV_ORDINAL&gt;</td>
<td>Set of allowed DV_ORDINAL values.</td>
<td></td>
</tr>
<tr>
<td>Invariants</td>
<td>Items_valid: not list.is_empty</td>
<td></td>
</tr>
</tbody>
</table>

5.3.2 C_DV_QUANTITY Class Definition

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_DV_QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constrain instances of DV_QUANTITY.</td>
</tr>
<tr>
<td>Inherit</td>
<td>C_DOMAIN_TYPE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Signature</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1..1 list: List&lt;C_QUANTITY_ITEM&gt;</td>
<td>List of value/units pairs.</td>
<td></td>
</tr>
<tr>
<td>0..1 property: DV_CODED_TEXT</td>
<td>Optional constraint on units property</td>
<td></td>
</tr>
<tr>
<td>Invariants</td>
<td>Items_valid: not list.is_empty</td>
<td></td>
</tr>
</tbody>
</table>
### 5.3.3 C_QUANTITY_ITEM Class Definition

<table>
<thead>
<tr>
<th>CLASS</th>
<th>C_QUANTITY_ITEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Constrain instances of <code>DV_QUANTITY</code>.</td>
</tr>
<tr>
<td>Inherit</td>
<td><code>C_DOMAIN_TYPE</code></td>
</tr>
<tr>
<td>Attributes</td>
<td>Signature</td>
</tr>
<tr>
<td></td>
<td><code>magnitude: Interval&lt;Real&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>units: C_STRING</code></td>
</tr>
<tr>
<td>Invariants</td>
<td><code>units_valid: units /= Void and not units.is_empty</code></td>
</tr>
</tbody>
</table>
END OF DOCUMENT
Introduction

This page provides access to a tool-based UML expression of the openEHR models. Two variants are available:

- **Browsable** - good for online browsing and printing of single diagrams, classes etc.
- **Printable** - good for printing the entire model (including diagrams), browsing in a serial fashion, and locating classes by name.

The Dictionary view from the browsable form provides text definitions of all classes and attributes.

The model is expressed in MagicDraw 9.5, and is available [here](#) in the standard zipped XML form used by MagicDraw.

Please report errors in these models to the openEHR webmaster.

Status

Summary

The models shown here are derived from the official openEHR Release-1.1 candidate (contains a small number of changes with respect to Release-1.0. PDF specifications). The latter should always be treated as being correct. The current status of the UML published here is draft. Various sources of difference from the official specifications exist:

- they contain some shortcomings due to using a UML 1.4 tool
- limitations in UML itself to correctly represent object-oriented semantics occasionally affect the models
- may contain human errors

Please report errors in these models with respect to the online specifications to the openEHR webmaster.

Detail

The UML models available from this page are derived from the official online PDF documentation. Although the UML models here are tool-produced there are a number of limitations of the tool (and these seem to be common with most tools on the market today) that have affected the fidelity of the representation in various small ways:

1. List<T>, Set<T>, Array<T>, Hash<T,K>, Interval<T> were all explicitly modelled, since they are not directly supported in UML. All uses of these "assumed types" were made though new Primitive types in MD such as List<String>, employing Binding Dependencies.
2. Associations are shown with far-end role names and Navigation arrows.

3. Constraint statements are syntactically very similar to the syntax used in the specifications, and nearly always pass OCL syntax checks. Differences are:

- Class constraints use the Inv: keyword followed by an OCL-compliant expression
- Function constraints use Pre: instead of require and Post: instead of ensure, or Inv:
- Double quotes are changed to single quotes
- Embedded spaces inside quoted strings used as parameters are changed to underscore
- Multiple clauses inside one Constraint statement are supported using separate Inv: or Pre: or Post: keywords
- Class constraints have names, Function constraints often do not.

In addition, there are imperfections in the model due to the fact that we have used version 9.5 MagicDraw, which does not support UML 2.0. The XMI output of the current tool is at version 1.0. Version 10 of the tool does support UML 2.0 and XMI 2, but upgrading is a non-trivial exercise, both of the model itself (semantics of UML 2 are quite different in some places) and of the publishing process we use to generate the online documentation.

Lastly, the data entry work of putting the specifications into the tool of course carries the possibility of error.

Future

In the medium term, the following things will probably happen:

- the models will be migrated from MagicDraw 9.5 to MagicDraw 10
- the publishing scripts (currently XSLT) will be migrated to MagicDraw 10 (Python)
- due to this migration, an XMI 2 expression of the model will become available

Acknowledgements

This Herculean task has been carried out by David Lloyd, from CHIME, UCL in London, for the openEHR Foundation.
**openEHR Terminology**

**Introduction**

This page provides access to the XML computable expression of the openEHR terminology, and a set of tools to support the use of this terminology in a variety of contexts including the Archetype Editor. The openEHR terminology is an adjunct to the openEHR reference and archetype models, and provides code sets and vocabularies needed by the models which are language independent. It is not a "real-world" ontology of any kind - i.e. it does not contain representation of facts in areas such as diseases or biochemistry - this is the job of much larger terminologies such as ICDx, ICPC, SNOMED-CT and so on.

The file contains:

- Concepts used in the reference model that are DV_CODED_TEXTs and are specified as coming from the openEHR Terminology. These concepts are grouped and have a group identifier. The group identifier can be found in the terminology tool provided from this site.
- Concepts that are represented in the user interface of the archetype editor, including labels, menus and dialog box messages.
- Concepts that are properties (e.g. mass, area) which restrict the set of units available in a DV_QUANTITY - this enables translation of these concepts.
- Language codes for use with openEHR (ISO).
- Territory codes for use with openEHR (ISO).
- A set of terminology identifiers used by UMLS for linking archetype terms to terminologies.

It does, presently, include the translations of all languages - this will be handled in separate files in future.

**File design**

The concepts are provided in a file called Terminology.xml with an associated schema, Terminology.xsd. The schema is an XML schema - documentation of this file is provided. The file and schema are provided here.

- Terminology.xml (200K)
- Terminology.xsd (7K)

**Tools**

Tools for manipulating the terminology have been developed for the openEHR Foundation by Ocean Informatics. Two tools are provided to manage this small terminology. Please note that the Terminology tool is provided for interest - any wanting to translate or alter this termset needs to contact Ocean Informatics to ensure the upgrade is coordinated. The Translation tool can be used independently, submitting the translated Terminology.xml file to Ocean Informatics for incorporation into the main file. (As explained above, the language specific files will be distributed separately in future). The files are:

1. The openEHR Terminology tool which allows the user to expand the terminology, add groups and edit any concepts in any language. This tool includes the translation tool.
2. The openEHR Translation tool which allows the user to translate the terminology to a new language. This can display all terms, or only those that are requiring translation.
To download the files please click on the links below. The tools are provided as .Zip files, but are actually .exe's - they need to be saved and run in the same directory as the Terminology.xml file.

- The Translation tool
- The Terminology tool

Work to be done

The current design of this small terminology is adequate for the purpose but there are a lot of potential improvements to this design. The most critical of these is to separate the language specific terms from the master concepts and groups. The current tools are written in VB.Net by Ocean Informatics - the source code of these applications can be made available to those seeking to improve this terminology. This terminology is presently being incorporated into a CTS compatible terminology service.

Acknowledgements

This work has been carried out by Ocean Informatics for the openEHR Foundation.
**openEHR XML-schemas**

---

**Introduction**

This page provides access to the W3C XML schemas for the openEHR reference model (RM). Separate schemas have been created, as follows:

- **Basic types** - includes data types and identification. ([xsd file](http://svn.openehr.org/specification/TRUNK/publishing/its/XML-schema/index.html (1 of 2))
- **Structure** - data_structures and generic. ([xsd file](http://svn.openehr.org/specification/TRUNK/publishing/its/XML-schema/index.html (1 of 2))
- **Content** - item, section, entry and all subtypes. ([xsd file](http://svn.openehr.org/specification/TRUNK/publishing/its/XML-schema/index.html (1 of 2))
- **Composition** - composition and event context. ([xsd file](http://svn.openehr.org/specification/TRUNK/publishing/its/XML-schema/index.html (1 of 2))

**Mapping Rules**

The following mapping rules were used in order to create the schema from the reference model.

**Basic type mappings**

The basic types utilise the internal XML simpleTypes in accordance with the assumed types section in the Support IM. Some changes have been made to the reference model to accommodate these standards.

<table>
<thead>
<tr>
<th>Reference model type</th>
<th>XML Schema complexType</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV_DATE_TIME</td>
<td></td>
</tr>
<tr>
<td>DV_DATE</td>
<td></td>
</tr>
<tr>
<td>DV_TIME</td>
<td></td>
</tr>
<tr>
<td>DV_PARTIAL_DATE</td>
<td></td>
</tr>
<tr>
<td>DV_PARTIAL_TIME</td>
<td></td>
</tr>
<tr>
<td>DV_PARTIAL_DATE_TIME</td>
<td></td>
</tr>
<tr>
<td>DV_DURATION</td>
<td></td>
</tr>
</tbody>
</table>

The value attribute is of xs:dateTime type based on ISO 8601

**Generic types**

The openEHR reference model makes use of generic types - including DV_INTERVAL<T> (Data types IM) and VERSION<T> (Common IM) for example. As generic types are now available in C# and Java this is warranted for implementation specifications. Generic types are not available in XML Schema, so specific types prefixed with 'XS_' have been created to provide the same constraints.

<table>
<thead>
<tr>
<th>Reference model type</th>
<th>XML Schema complexType</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV_INTERVAL&lt;DV_ORDINAL&gt;</td>
<td>XS_DV_INTERVAL_ORDINAL</td>
</tr>
</tbody>
</table>
### openEHR XML Schemas

<table>
<thead>
<tr>
<th>Reference model type</th>
<th>XML Schema complexType</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHETYPED</td>
<td>Removed from LOCATABLE</td>
</tr>
</tbody>
</table>

**Removed attributes**

Some attributes have been removed for the moment as the lack of surety about implementation of this feature is not well understood.

**Reference model type | XML Schema complexType**

| ARCHETYPED               | Removed from LOCATABLE |

**Removed from LOCATABLE**

**Added attributes**

All added attributes have current change requests outstanding. Any that are not accepted within the 3 month public review will be added to this table.

**Reference model type | XML Schema complexType**

| To Be Determined         |                         |

**To Be Determined**

---

**Acknowledgements**

This work was carried out for the openEHR Foundation by Ocean Informatics.

---

$LastChangedDate: 2006-02-15 05:57:48 +0930 (Wed, 15 Feb 2006) $

$LastChangedRevision$
openEHR Reference Model Java ITS

Revision: 1.5
Pages: 18

Author: Rong Chen
rong@acode.se

ACODE HB, Sweden
http://www.acode.se
## Revision History

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Details</th>
<th>Who</th>
<th>Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>Added openEHR image</td>
<td>R Chen</td>
<td>02/09/06</td>
</tr>
<tr>
<td>1.4</td>
<td>Incorporated T Beale's comments</td>
<td>R Chen</td>
<td>01/12/06</td>
</tr>
<tr>
<td>1.3</td>
<td>Some minor updates</td>
<td>R Chen</td>
<td>12/06/05</td>
</tr>
<tr>
<td>1.2</td>
<td>Added Demographic and EHR Extract information model</td>
<td>R Chen</td>
<td>03/13/05</td>
</tr>
<tr>
<td>1.1</td>
<td>Switched target JDK from 1.4 to 5.0</td>
<td>R Chen</td>
<td>10/15/04</td>
</tr>
<tr>
<td>1.0.2</td>
<td>“DV_” prefix kept and renamed to “Dv”</td>
<td>R Chen</td>
<td>09/07/04</td>
</tr>
<tr>
<td>1.0.1</td>
<td>Added “Implementations from Java API”, separated “immutability from value object” and some minor corrections</td>
<td>R Chen</td>
<td>08/27/04</td>
</tr>
<tr>
<td>1.0.0</td>
<td>Initial writing</td>
<td>R Chen</td>
<td>08/11/04</td>
</tr>
</tbody>
</table>

## Acknowledgments

Thanks to Thomas Beale and others from openEHR for the original work, and Thomas Beale particularly for valuable discussions and the initial invitation to write this document.

Thanks to Göran Pestana, my partner at ACODE HB Sweden for the initial review and support.
1 Introduction

1.1 Purpose

This document provides recommendations and guidelines for Java implementation of the openEHR Reference Model, which at the time of writing consists of the Reference Information Model and the Archetype Model. The Reference Information Model includes Data Types, Demographic, Common, Support, Data Structures, EHR and EHR Extract model.

1.2 Related Documents

- The openEHR Data Types Information Model
- The openEHR Common Information Model
- The openEHR Demographic Information Model
- The openEHR Support Information Model
- The openEHR Data Structures Information Model
- The openEHR EHR Information Model
- The openEHR EHR Extract Information Model
- The openEHR Archetype Object Model
- The openEHR Archetype Definition Language
- The openEHR Archetype Profile

2 Background

2.1 Scope

This document is based on the work on a Java implementation of the openEHR Reference Model. The code has been released by ACODE under open source license and later adopted by the openEHR foundation as the reference Java implementation.

The recommendations presented here only apply to perspective Java implementation of the openEHR Reference Model. No presumption about usage of the reference model, e.g. used within
desktop application or server-side component has been made. The idea is that the reference model implementation should be generic enough to be used in different application context. There is no design choice made for specific persistence or presentation solution either. Since in an archetype-based system, the reference model probably should not have any dependency on either the presentation layer or the persistence layer.

2.2 Guideline Criteria

The openEHR Reference Model is designed to be programming language independent. It has been developed according to the UM semantics, and implemented thus validated in Eiffel programming language.

There are a number of things one needs to bear in mind when implements the openEHR Reference Model in Java. Some of them are cosmetic issues, like naming conventions, some of them are more profound, for example, Design By Contract in Eiffel, which is heavily used to reinforce the reference model and but not supported natively in Java. In general, the goal is to implement the openEHR model in Java as faithful to the original specifications as possible and at same time keep the Java look-and-feel.

1. Java implementation must be able to present the information that can be presented by the original class model in the specifications.

2. It should look similar to the original model in terms of class names, attribute names and method signatures, so that mapping between the Java implementation and the original model can be easily made.

3. Java implementation should look like Java. That is it should follow Java standards, naming conventions and idioms, make use of both built-in and well-known APIs from third parties instead of re-inventing the wheel.
2.3 Java Platform

The Java platform targeted here is Java 2 Platform, Standard Edition, v5.0 mainly because of newly added support of generic types.

The Enterprise Edition is not required to implement the reference model, but it will be very useful in implementing the Service Model. The 'assert' keyword added since J2SE v 1.4 could be useful to implement DBC like pre-conditions (see more in DBC section).

3. Assumed Types

The list of assumed types are quoted from openEHR Support Information Model document (support_im-1_1.pdf). The mapping to Java types[10] is listed below:

<table>
<thead>
<tr>
<th>Assumed Type</th>
<th>Java Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>Object</td>
<td>super class of all Java types</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean, Boolean</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>Character</td>
<td>char, Character</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>Integer</td>
<td>int, Integer</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>Integer_64</td>
<td>long, Long</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>Real</td>
<td>float, Float</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>Double</td>
<td>double, Double</td>
<td>primitive type or class</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
<td>Unicode is natively supported</td>
</tr>
<tr>
<td>Container</td>
<td>java.util.Collection</td>
<td>In Java, Container is</td>
</tr>
</tbody>
</table>
### Assumed Type, Java Type, Comment

<table>
<thead>
<tr>
<th>Assumed Type</th>
<th>Java Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array&lt;T&gt;</td>
<td>T[]</td>
<td>not parent of arrays</td>
</tr>
<tr>
<td>List&lt;T&gt;</td>
<td>java.util.List</td>
<td>supported as first-class objects</td>
</tr>
<tr>
<td>Set&lt;T&gt;</td>
<td>java.util.Set</td>
<td>subclass of Collection</td>
</tr>
<tr>
<td>Bag&lt;T&gt;</td>
<td>org.apache.commons.collections.bag</td>
<td>subclass of Collection</td>
</tr>
<tr>
<td></td>
<td>not supported</td>
<td>from Apache Software Foundation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Appendix.B for sample implementation</td>
</tr>
</tbody>
</table>

### 4. Naming Convention

In general, naming of packages, classes, methods and attributes should follow the Java Naming Convention[11]

- **packages**
  
  DATA_TYPES.BASIC -> datatypes.basic

  In reality, one would put a full domain name in the path to make it unique, something like se.acode.openehr.datatypes.basic

- **classes**
  
  e.g. DATA_VALUE -> DataValue

  For all the subclasses of DataValue in datatypes package, “DV_” prefix of the class name is kept but renamed to “Dv” to follow the Java naming convention.

- **fields**
  
  e.g. calendar_alignment -> calendarAlignment

- **methods**
  
  e.g. is_strictly_comparable_to() -> isStrictlyComparableTo()

- **accessors**
  
  For fields (attributes) that are declared by the specification, they should be implemented as private fields and public accessors(getters) should be provided to access them.
5. Generic Types

Generics of openEHR Reference Model is implemented directly by Java generics, which is introduced since Java 5.0.

6. Design By Contract

There are several Java implementations of Design By Contract, e.g. iContract, which could be used in Java environment. The recommendations here could be used without involvement of any Java implementation of DBC.

1. Pre-conditions

Pre-conditions for public methods that are related to parameters should be implemented as parameter validation and IllegalArgumentException or its subclass should be thrown when pre-conditions can not be satisfied. For non-public methods, pre-conditions can be implemented as assertions using keyword 'assert', which is added into Java since Java1.4. It worths to mention that pre-conditions for public methods that belong to parameter validation should not be implemented as assertions because these contracts must be obeyed whether assertions are enabled or not. Also bad parameters to the public methods should result in runtime exception instead of assertion failure. See Appendix.B for example.

2. Post-conditions

Unit testing can be used to verify the result after execution of methods. Compared with DBC post-conditions, unit testing tests the result of specific condition by supplying known test data, while DBC post-conditions are more general. Because of that, it is probably easier to write unit tests than post-conditions[3].

3. Invariants

Invariants can be implemented as parameter validation in the constructors if the object is immutable. Since the internal state does not change during the lifetime of any immutable object.
Otherwise invariants can also be implemented as methods and put at strategic places.

7. Value Object

Most of the datatypes classes are essentially Value Objects\([2]\), whose purpose is to pass values around. The equality of Value Object is not based on the identity, but based on the values of contained fields. Therefore, override both equals() and hashCode() for all Value Objects. Use the values of all relevant fields in equals() and hashCode() and obey general contract of both methods. See Appendix.B for example

8. Immutable Object

Most of classes in the reference model are good candidates of immutable objects. Immutable objects are easier to design and implement than mutable ones. They are also easier to use and much safer. To make a class immutable, follow the following rules\([15]\):

1. Make all fields private and final
2. Make the class final
3. Only provide accessors methods to fields. Do not provide any mutators (setters).
4. Make defensive copies in constructors and accessors when fields refer to mutable objects, e.g. instances of Collection and its subclasses. For collection instances, one could use java.util.Collections.unmodifiableXXX() methods to get unmodifiable view of the collection. See Appendix.B for example

9. Exceptions

Parameter validation in public methods and constructors should throw IllegalArgumentException or its subclass if the specified preconditions can not be satisfied. It is important to throw Exceptions
that subclass RuntimeException instead of checked Exceptions to indicate that the failure is unrecoverable.

See Appendix.B for example

10. Multiple Inheritance

Multiple inheritance is not directly supported in Java. Luckily, there are few classes in the openEHR reference model that inherit multiple super classes. When required, one of the parent classes can be chosen to be implemented as the super class in Java, the others should then be implemented as fields. The criterion for choosing the parent class is based on its substitutability. For example, all subtypes of DataValue should be substitutable for DataValue, so any class that inherits DataValue really has to inherit it. The class datatypes.quantity.Interval is the only class in the entire Data Types model that multiply inherits - it also inherits assumed type Interval - but this is for implementation - no substitutability is needed. So it should inherit DataValue and include an instance of assumed type Interval as a field.

11. Abstract attributes

Abstract attributes is not supported in Java. The solution is declare it as abstract method and return the required type, then the subclass will simply implement it and return more specialized types.

12. Operator Overloading

Operator overloading is not supported in Java, therefore it should be implemented as method in the class. e.g. infix ‘<’ implemented as compareTo() of interface Comparable infix ‘+’ implemented as add() method infix ‘-’ implemented as substract() method prefix ‘-’ implemented as negate() method
13. Index

Index values of array and List always starts with 0 in Java. For example, the Java implementation of ithItem() method in ItemList takes index starting from 0.

14. String Representation

All classes should override the method toString() to provide human readable information. When appropriate, toString() should also present all interesting information contained in the object. See Appendix.B for example.

14.1 XML representation

XML based string representation can be useful and it should probably be required to be implemented for all Data Types classes. The XML string should include all non-calculated values from the object so that it would be possible to reconstruct the object later by parsing the XML string. The format of the XML string needs to be standardized by the openEHR, but one possibility is to use the attribute name as element name thus generating and parsing XML string can be automated by using Java Reflection API.

15. Comparable

Comparable interface should be implemented if instance of the class has natural order, e.g. datatypes.quantity.Ordered. By implementing Comparable interface, one could take advantage of many generic algorithms and collection implementations provided by the Java platform[15].

16. Implementations from Java API

There are existing classes from Java platform already provide concrete implementation of classes defined by openEHR.

- java.util.Calendar for implementation of classes in datatypes.quantity.datetime package, more specifically DvDate, DvTime and DvDateTime are simply thin wrappers of java.util.Calendar
17. External APIs

There are well known external open source Java APIs that can provide solid implementation or test framework.

1. JUnit, a unit testing framework for Java.
   http://junit.org

2. Jakarta commons-lang, provides highly reusable static utility methods, chiefly concerned with adding value to java.lang and other standard core classes.
   http://jakarta.apache.org/commons/lang/

3. Jakarta commons-collections, contains implementations, enhancements and utilities that complements the Java Collections Framework
   http://jakarta.apache.org/commons/collections/

4. JDOM, XML API for manipulate XML in Java way,
   http://www.jdom.org
Appendix

A. References

A.1 General

2. Fowler M, Patterns of Enterprise Application Architecture Addison Wesley 2003

A.2 openEHR

4. Beale T et al, Design Principles for the EHR
5. Beale T et al, openEHR Data Types Information Model
6. Beale T et al, openEHR Support Information Model
7. Beale T et al, openEHR Common Information Model
8. Beale T et al, openEHR Data Structures Information Model
9. Beale T et al, openEHR EHR Information Model
10. Beale T et al, openEHR Archetype Object Model
11. Beale T et al, openEHR Archetype Definition Language

A.3 Java

B. Sample implementation of Interval

```java
package se.acode.openehr.support.basic;

import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;

/**
 * Interval of comparable items. Instances of this class
 * are immutable.
 *
 * @author Rong Chen
 * @version 1.0
 */
public final class Interval {

    /**
     * Constructs an Interval
     *
     * @param lower null if unbounded
     * @param upper null if unbounded
     * @throws IllegalArgumentException if lower > upper
     */
    public Interval(Comparable lower, Comparable upper) {
        if (lower != null && upper != null && upper.compareTo(lower) < 0) {
            throw new IllegalArgumentException("lower > upper");
        }
        this.lower = lower;
        this.upper = upper;
    }

    /**
     * Returns the lower boundary of this Interval
     * @return null if unbounded
     */
    public Comparable getLower() {
        return lower;
    }

    /**
     * Returns the upper boundary of this Interval
     * @return null if unbounded
     */
    public Comparable getUpper() {
        return upper;
    }
}
```
public Comparable getUpper() {
    return upper;
}

/**
 * Returns true if lower boundary open
 *
 * @return true if has lower boundary
 */
public boolean isLowerUnbounded() {
    return lower == null;
}

/**
 * Returns true if upper boundary open
 *
 * @return true if has upper boundary
 */
public boolean isUpperUnbounded() {
    return upper == null;
}

/**
 * Returns true if (lower >= value and value <= upper)
 *
 * @param value to compare to
 * @return true if given value is included in this Interval
 * @throws IllegalArgumentException if value is null
 */
public boolean has(Comparable value) {
    if (value == null) {
        throw new IllegalArgumentException("null value");
    }

    return (lower == null || value.compareTo(lower) >= 0) && (upper == null || value.compareTo(upper) <= 0);
}

/**
 * Equals if two Intervals have same values for lower and upper boundaries
 *
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Interval)) return false;
    final Interval interval = (Interval) o;
    return new EqualsBuilder()
        .append(upper, interval.upper)
        .append(lower, interval.lower)
        .isEquals();
}

/**
 * Return a hash code of this Interval
 *
 * @return hash code
 */
public int hashCode() {
    return new HashCodeBuilder()
        .append(upper)
        .append(lower)
        .toHashCode();
}

/**
 * Return string representation of this Interval. The string
 * consists of both lower and upper boundary, if any of them
 * is not specified, "unbounded" is provided.
 *
 * @return string representation
 */
public String toString() {
    StringBuffer buf = new StringBuffer("Interval [lower: ");
    buf.append(lower == null ? "unbounded" : lower);
    buf.append(" , upper: ");
    buf.append(upper == null ? "unbounded" : upper);
    buf.append(" ]");
    return buf.toString();
}
C. Sample Code of unit testing of Interval

/**
 * IntervalTest
 *
 * @author Rong Chen
 * @version 1.0
 */
package se.acode.openehr.support.basic;

import junit.framework.TestCase;

public class IntervalTest extends TestCase {

    public IntervalTest(String test) {
        super(test);
    }

    /**
     * The fixture set up called before every test method.
     */
    protected void setUp() throws Exception {
    }

    /**
     * The fixture clean up called after every test method.
     */
    protected void tearDown() throws Exception {
    }

    public void testConstructor() throws Exception {
        try {
            new Interval(new Integer(10), new Integer(1));
            fail("should throw illegal argument exception");
        } catch (Exception ignored) {
        }
    }

    public void testHas() throws Exception {

array of { lower(0:unbounded), upper(0:unbounded),
  testValue, expected (1:true, 0:false) }

int[][] data = {
  {1, 8, 2, 1},
  {1, 8, 1, 1},
  {1, 8, 8, 1},
  {1, 8, 0, 0},
  {1, 8, 9, 0},
  {0, 8, 4, 1},
  {0, 8, -1, 1},
  {0, 8, 9, 0},
  {1, 0, 4, 1},
  {1, 0, 1, 1},
  {1, 0, -1, 0}
};

for (int i = 0; i < data.length; i++) {
  Interval iv = new Interval(popInt(data[i][0]),
    popInt(data[i][1]));
  boolean actual = iv.has(new Integer(data[i][2]));
  boolean expected = data[i][3] == 1;
  assertTrue("failed at " + testString(data[i]),
    actual == expected);
}

private Integer popInt(int value) {
  if (value == 0) {
    return null;
  }
  return new Integer(value);
}

private String testString(int[] row) {
  return "(" + row[0] + ", " + row[1] + ") has " +
    row[2] + ": " + (row[3] == 1);
}

public void testToString() throws Exception {
  int[][] data = {
    {10, 100}, {0, 100}, {-20, 0}, {0, 0}
  };
  String[] expected = {
    "Interval [lower: 10, upper: 100]",
  };
}
"Interval [lower: unbounded, upper: 100]",
"Interval [lower: -20, upper: unbounded]",
"Interval [lower: unbounded, upper: unbounded]"

};
for (int i = 0; i < data.length; i++) {
    assertEquals(expected[ i ] + " expected",
        expected[ i ],
        new Interval(popInt(data[ i ][ 0 ]),
        popInt(data[ i ][ 1 ])).toString());
}

public void testEquals() throws Exception {
    Interval interval = new Interval(new Integer(-1), new
        Integer(10));
    Interval interval2 = new Interval(new Integer(-1), new
        Integer(10));
    assertEquals(interval, interval2);

    // not equals expected
    int[][] data = {
        (-1, 9), {2, 10}, {0, 10}, {-1, 0}, {0, 0}
    };

    for(int i = 0; i < data.length; i++) {
        interval2 = new Interval(popInt(data[i][0]),
            popInt(data[i][1]));
        assertFalse(interval2.toString(),
            interval.equals(interval2));
        assertFalse(interval2.toString(),
            interval2.equals(interval));
    }
}
}