<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Reference model manager</td>
<td>35</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Importing a new reference model</td>
<td>36</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Reference model documentation</td>
<td>38</td>
</tr>
<tr>
<td>6.1.2.1</td>
<td>Mindmap generation</td>
<td>41</td>
</tr>
<tr>
<td>6.2</td>
<td>Semantic pattern manager</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>Data source manager</td>
<td>42</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Data source import</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Working with archetypes</td>
<td>46</td>
</tr>
<tr>
<td>7.1</td>
<td>Creating a new archetype</td>
<td>46</td>
</tr>
<tr>
<td>7.2</td>
<td>Opening an existing archetype</td>
<td>47</td>
</tr>
<tr>
<td>7.3</td>
<td>Specializing an archetype</td>
<td>47</td>
</tr>
<tr>
<td>7.4</td>
<td>Saving an archetype</td>
<td>48</td>
</tr>
<tr>
<td>7.5</td>
<td>Archetype import</td>
<td>48</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Archetype Repository</td>
<td>48</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Open XML archetype</td>
<td>49</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Open Lkr file</td>
<td>49</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Open OET template</td>
<td>49</td>
</tr>
<tr>
<td>7.5.5</td>
<td>Open OPT template</td>
<td>49</td>
</tr>
<tr>
<td>7.5.6</td>
<td>Import FHIR</td>
<td>49</td>
</tr>
<tr>
<td>7.5.7</td>
<td>Import ADL2 archetype</td>
<td>50</td>
</tr>
<tr>
<td>7.6</td>
<td>Archetype export</td>
<td>50</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Export mindmap</td>
<td>50</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Export Excel</td>
<td>51</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Export XML archetype</td>
<td>51</td>
</tr>
<tr>
<td>7.6.4</td>
<td>Export JSON Schema</td>
<td>51</td>
</tr>
<tr>
<td>7.6.5</td>
<td>Export ADL for openEHR editor/Template designer</td>
<td>51</td>
</tr>
<tr>
<td>7.6.6</td>
<td>Export Operational Template (OPT)</td>
<td>51</td>
</tr>
<tr>
<td>7.6.7</td>
<td>Export Canonical Schema</td>
<td>51</td>
</tr>
<tr>
<td>7.6.8</td>
<td>Export LKR</td>
<td>52</td>
</tr>
<tr>
<td>7.7</td>
<td>Share archetype</td>
<td>52</td>
</tr>
<tr>
<td>7.8</td>
<td>Archetype printing</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>Edition of archetypes</td>
<td>53</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8.1</td>
<td>Archetype metadata</td>
<td>53</td>
</tr>
<tr>
<td>8.2</td>
<td>Constraint definition</td>
<td>53</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Add and remove constraints</td>
<td>53</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Object nodes</td>
<td>54</td>
</tr>
<tr>
<td>8.2.2.1</td>
<td>Occurrences</td>
<td>54</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Attribute nodes</td>
<td>55</td>
</tr>
<tr>
<td>8.2.3.1</td>
<td>Existence</td>
<td>55</td>
</tr>
<tr>
<td>8.2.3.2</td>
<td>Cardinality</td>
<td>55</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Primitive types</td>
<td>55</td>
</tr>
<tr>
<td>8.2.4.1</td>
<td>String type</td>
<td>55</td>
</tr>
<tr>
<td>8.2.4.2</td>
<td>Integer type</td>
<td>56</td>
</tr>
<tr>
<td>8.2.4.3</td>
<td>Real (double) type</td>
<td>56</td>
</tr>
<tr>
<td>8.2.4.4</td>
<td>Boolean type</td>
<td>57</td>
</tr>
<tr>
<td>8.2.4.5</td>
<td>Date-time type</td>
<td>57</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Special nodes</td>
<td>57</td>
</tr>
<tr>
<td>8.2.5.1</td>
<td>Internal references</td>
<td>57</td>
</tr>
<tr>
<td>8.2.5.2</td>
<td>Archetype slots</td>
<td>58</td>
</tr>
<tr>
<td>8.2.5.3</td>
<td>Constraint references</td>
<td>58</td>
</tr>
<tr>
<td>8.2.5.4</td>
<td>Domain types</td>
<td>59</td>
</tr>
<tr>
<td>8.3</td>
<td>Archetype language translations</td>
<td>59</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Language management</td>
<td>60</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Translating an archetype</td>
<td>60</td>
</tr>
<tr>
<td>8.4</td>
<td>Archetype ontology definition</td>
<td>60</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Term definition</td>
<td>61</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Constraint definition</td>
<td>61</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Term binding</td>
<td>61</td>
</tr>
<tr>
<td>8.4.4</td>
<td>Constraint binding</td>
<td>61</td>
</tr>
<tr>
<td>8.5</td>
<td>Advanced edition</td>
<td>62</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Semantic patterns</td>
<td>62</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Edition of archetypes without reference model</td>
<td>64</td>
</tr>
<tr>
<td>8.5.3</td>
<td>ADL edition</td>
<td>64</td>
</tr>
<tr>
<td>8.6</td>
<td>Archetype validation</td>
<td>64</td>
</tr>
</tbody>
</table>
8.7 Rules definition
  8.7.1 Variables
  8.7.2 Rules
8.8 Advanced utilities
  8.8.1 Export XML instance with options
  8.8.2 Schematron
  8.8.3 Implementation guide
  8.8.4 Implementation guide from directory
  8.8.5 Export HTML report
  8.8.6 Parse XML instance
  8.8.7 Generate FHIR DSTU profile
  8.8.8 Generate Natural language rules
  8.8.9 Extract entities as slots
  8.8.10 Transform openEHR to ISO13606
  8.8.11 FHIR
    8.8.11.1 Transform FHIR to openEHR
    8.8.11.2 Transform openEHR to FHIR Observation
    8.8.11.3 Transform openEHR to FHIR Questionnaire
    8.8.11.4 Transform archetype to FHIR Logical Model
9 Mapping of archetypes
  9.1 Transformation archetypes
  9.2 Mapping of archetypes to data sources
  9.3 Archetype mapping interface
    9.3.1 Archetype mapping tree
    9.3.2 Archetype mapping table
  9.4 Mapping definition
    9.4.1 Data source selection
    9.4.2 Mapping expression syntax
    9.4.3 Common operators
    9.4.4 Math functions
    9.4.5 String functions
    9.4.6 Time functions
9.4.7 Ontological functions 100
9.4.8 Aggregation functions 104
9.4.9 Constraint values 107
9.4.10 Data source access 107
9.5 Advanced mapping 108
  9.5.1 Object mapping 108
  9.5.2 Virtual node mapping 110
  9.5.3 Mapping reuse 110
    9.5.3.1 Internal references 111
    9.5.3.2 Archetype slots 111
10 Generation of archetype transformation programs 112
  10.1 Selection of covers 115
  10.2 Skolem function edition 116
  10.3 Testing the archetype transformation program 116
  10.4 Exporting the archetype transformation program 116
  10.5 Using the archetype transformation program outside LinkEHR Studio 116
11 Troubleshooting 117
Annex 1. The ADL language 118
Annex 2. CEN/ISO 13606 specific editor 119
  o Choosing a root entity for a CEN EN13606 archetype 119
  o Editing CEN EN13606 archetypes with LinkEHR 121
    • CEN EN13606 Datatypes 123
Annex 3. OpenEHR specific editor 125
Annex 4. HL7 CDA specific editor 135
  o Choosing a root entity for a HL7 CDA archetype 135
  o Editing HL7 CDA archetypes with LinkEHR-Ed 136
  o HL7 CDA Datatypes 138
Annex 5. Licenses 140
About us

VeraTech for Health ([www.veratech.es](http://www.veratech.es)) is an innovative eHealth ICT SME located in Valencia, Spain, whose mission is to facilitate the access and reuse of relevant health data and information wherever and whenever it is needed, through advanced tools for integrations, standardisation, data mining, knowledge modelling, decision support and consulting services.

To get more information about us and our products visit:

[www.linkehr.com](http://www.linkehr.com)

[www.veratech.es](http://www.veratech.es)
About this document

This manual is part of the reference documentation of the LinkEHR Interoperability Platform. You will find more information about other LinkEHR modules in our webpage.

www.linkehr.com

This document uses the following symbols:

<table>
<thead>
<tr>
<th>(NO SYMBOL)</th>
<th>Functionality freely available in LinkEHR Studio demo version.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ST</strong></td>
<td>Functionality only available for the commercial version of LinkEHR Studio.</td>
</tr>
<tr>
<td><strong>i</strong></td>
<td>Additional information.</td>
</tr>
</tbody>
</table>
About LinkEHR Interoperability Platform

To obtain a unified and universal electronic health record (EHR) for each person is one of the most important objectives of health informatics. This EHR should include all the existing information related to a person from his birth to his death, independently of the place where the patient has received attention.

Resolving this problem requires interconnecting all the information systems and achieving an agreement about the format of the transmitted information. Not only the syntax is important, but also the meaning of the information, which assures a correct interpretation by human readers or computer systems.

The LinkEHR Interoperability Platform helps solving both problems.

The LinkEHR Interoperability platform

The LinkEHR Interoperability Platform (www.linkehr.com) allows creating an integrated and normalized view of the EHR of a patient whose data are distributed among heterogeneous information systems in three basic steps.

- **Data integration.** LinkEHR Integration gains access to your data sources and documents and integrates them in a virtual view built on-the-fly in XML format. It assures the highest degree of security and privacy for medical data.
- **Data normalization.** LinkEHR Studio is a tool that helps implementing the semantic interoperability of health information by allowing the edition of archetypes, and the standardization of existing data into a standard format according to those archetypes.
- **Model sharing.** LinkEHR Model Manager is a web application for the publication, management and governance of clinical information models and other semantic artefacts, including archetypes, templates, and their related documentation.
1 Introduction

1.1 LinkEHR Studio

LinkEHR Studio is a tool that helps to implement the semantic interoperability of health information. It helps you in two of the main tasks you will need to execute:

- **Edition of archetypes**, as a representation of the clinical models used in your EHR.
- **Standardization of existing data** into a format according to the archetypes and your selected EHR standard (openEHR, ISO 13606, HL7 CDA...).

LinkEHR Studio drastically reduces the time and efforts needed to adopt archetypes in existing EHR systems, from weeks or months to just days.

LinkEHR Studio includes a multi-model archetype editor that is focused on simplifying the archetype definition process but maintaining access to all the complex details when needed. It incorporates specific interfaces and functionalities to create, specialize and review archetypes. The tool can automatically generate from an archetype definition a mindmap of its structure, summary spreadsheets with the most relevant information of the archetype, sample forms that show how the archetype might look like in a final user interface, sample data instances and validators of existing data.

LinkEHR Studio supports multiple reference models and standards. Any information model, standard or proprietary, can be imported into the tool and be used to create archetypes based on it. Examples of information models already supported by LinkEHR Studio are ISO 13606, openEHR, HL7 CDA, HL7 FHIR, ASTM CCR and CDISC ODM.

LinkEHR Studio connects to LinkEHR Model Manager and to the openEHR CKM servers. At any moment, the user can search and download existing archetypes from those repositories and start editing them right away.

Once we have defined a set of archetypes, one question arises: **how can we use them in existing EHR systems and with legacy data?**. LinkEHR Studio allows the specification of mappings between archetypes and existing data sources (legacy data repositories). These mappings define how to obtain a data value for each attribute of the archetype from the data sources, performing data transformations if necessary. Transformation can be also defined between different standards. For example, you can transform ISO 13606 data into openEHR data and vice-versa, or you can reuse the information of a medication archetype to create a dispensation document.

Based on that mapping, LinkEHR Studio automatically generates a transformation script that is able to transform legacy data instances into normalized documents using standard XML technologies (XQuery).
1.2 What is an archetype?

The LinkEHR Interoperability Platform is based on the archetype concept. An archetype is a representation of a clinical information model. That means that it defines a particular
configuration of the reference model, with a specific meaning. For example, an archetype can represent the information structure that corresponds to a vital signs measurement, a laboratory test, or a complete family history report.

Example of an archetype of a Patient Summary report

Archetypes are built by combining and constraining the existent classes of the reference model. We define a structure for the classes, select the data types, and constrain the range of possible values that can be registered according to the archetype. Additionally, archetypes can be mapped to terminologies to provide a specific meaning for the information structure, thus making it semantically interoperable.

One powerful characteristic of archetypes is their reusability. Archetypes that are defined can afterwards be reused by combining them into more complex archetypes, by specializing them to fit particular scenarios of use, and by versioning them to align to new requirements.
2 Installation

2.1 Technical requirements

The basic technical requirements for installing and using LinkEHR are the following:

Hardware requirements

- Intel x86 or compatible processor at 1.5GHz
- 2GB RAM
- 500MB hard disk space

Software requirements

- Windows XP or upper
- Java VM 1.8 or upper
- Adobe Flash to visualize FreeMind files

2.2 Installation of LinkEHR

Java installation

LinkEHR is based on Java and needs a Java Virtual Machine installed on the system. To check if it is installed on the system type “java -version” on a Windows console (you can open the console by clicking Start ➔ Execute... and typing cmd.exe).

If Java VM is installed, the version number will be shown. LinkEHR requires Java VM 1.8 or upper. If it is not available, please proceed to install it through the www.java.com web page.

LinkEHR installation

You can obtain LinkEHR studio by downloading it from www.linkehr.com. LinkEHR is installed just by unzipping the downloaded file into any folder of your computer. Once unzipped, LinkEHR can be executed by double clicking the executable “LinkEHR.exe”

| ADDITIONAL INFORMATION |

LinkEHRStudio is available as a free download from www.linkehr.com

It includes all the functionality related to the edition of archetypes, but not the mapping and transformation of data functionalities. To gain access to the complete functionality of LinkEHR Studio you need to purchase a commercial license from VeraTech for Health www.veratech.es
### 2.3 Installation of the license

The LinkEHR Studio version that can be downloaded from [www.linkehr.com](http://www.linkehr.com) is a demo version, with limited functionality. Once you purchase a commercial version of LinkEHR Studio, you can upgrade the demo version to fulfil all the functionalities.

1. Obtain a valid LinkEHR license number at [www.veratech.es](http://www.veratech.es). It will be delivered as a *.lcfg (LinkEHR configuration) file.
2. Run LinkEHR and, in the menu bar, click Configuration → Import configuration.
3. In the file selection window, select the *.lcfg file and accept.
4. Close and restart LinkEHR for the license to take effect.

![Import configuration](image-url)
3 LinkEHR Studio in three steps

ADDITIONAL INFORMATION

In this section, we will use the Blood pressure example that you will find in the examples folder of your LinkEHR installation folder.

3.1 Open/Create and edit an archetype

When opening LinkEHR Studio for the first time, you will see the following empty environment:

In the menu bar, click Archetype Open or click the button

Navigate to the LinkEHR\examples folder and Select the archetype named CEN-EN13606-ENTRY.Blood_pressure.v1.adl. The archetype will be loaded into LinkEHR Studio.

ADDITIONAL INFORMATION
You can also create a new archetype as explained in Section 8.2.

The tree at the left shows the complete archetype structure. You can navigate and select any of its nodes. The related information is shown in the right side. For example, if you select the Systolic node in the archetype tree, you will see the details of that node. As shown in the following image, in that case you can change the occurrences, the name, description and terminological binding of the node.

**ADDITIONAL INFORMATION**

You can find detailed information about the applicable constraints of an archetype node in Section 9.4.
You can also inspect a complete view of the archetype in the form of a mind map by clicking the button

3.2 Map an archetype

Once the archetype definition is complete, we can map its information structure to an existing data structure. These mapping will be used by LinkEHR Studio to automatically generate a transformation program that will convert existing data in a proprietary or not normalized format into standard data according to the archetype definition.

To enter the mapping environment, click the button in the toolbar. The archetype tree will change to incorporate not only the structure of the archetype but also all the nodes that exist in the reference model that is being used as the basis for the archetype. Thus, you will be able to map and transform data not only corresponding to the archetype nodes, but also for all contextual information represented by default by the reference model.
Archetype nodes that have been already mapped are rendered in green. Mandatory nodes that have not been mapped are rendered in red. Optional nodes that have not been mapped are rendered in black. A minimal mapping for an archetype requires that the archetype root node is rendered in green.

Mappings are defined in the archetype tree leaf nodes, which always correspond to primitive data types. Navigate to a leaf node, select it, and you will see the mapping edition table at the right side. You can create or edit a mapping by double clicking a cell of the mapping table. A new window will be opened that allows defining a condition or mapping expression between the archetype node and a data source information structure.

**ADDITIONAL INFORMATION**

You can find detailed information about importing a data source in Section 13.4.

**ADDITIONAL INFORMATION**

You can find detailed information about mapping definition in Section 14.
3.3 Compile, test and deploy a transformation program

Once a mapping is complete (green root node of the archetype tree), we can generate the data transformation program by clicking the button \( \rightarrow \). A new window will be opened, which includes the mapping summary (an archetype tree with only those nodes that have been mapped), the preview of the generated XQuery transformation program, the possibility to test the transformation, and the results export options.
ADDITIONAL INFORMATION

You can find detailed information about exporting a transformation program in Section 15.

To test that the transformation works correctly, click the **Test XQuery** button. You will be asked to select an XML file that corresponds to the original data format. In our example, select the file `legacy_data_BloodPressure.xml`. The result of executing the transformation will be shown in a new window. You can choose to see the resulting XML that has been transformed, a JSON representation of it, or a readable representation in HTML format.
4 LinkEHR Studio environment

4.1 Overall description

LinkEHR Studio is organized in three main areas:

- **Archetype tree.** When an archetype is opened, its complete structure is shown here in a tree form. You can navigate the archetype structure and select any of its nodes to edit its properties and constraints. From the archetype tree, you have also access to the archetype metadata, to its language translations, and the ontology section of the archetype. More info in Section 4.4.

- **Constraints definition.** In this area, you can edit the main properties and constraints of the archetype. The form that is presented will vary in function of the selected node of the archetype tree. More info in Section 4.5

- **Additional information.** In this area, additional information about the tool status, error messages, and information about the selected nodes are shown. Additionally, it provides quick access to the Terminology library included in LinkEHR Studio. More info in Section 4.6

4.2 Menu bar

LinkEHR Studio menu bar provides access to specific and advanced functionalities.
- **Archetype.** Functionality related to the edited archetype: create new archetype, open, specialize, import, export, share and print options.
- **Edit.** Provides access to the list of possible new nodes that can be added at some point of the archetype.
- **Managers.** Provides access to the reference model manager, the semantic pattern manager and the datasource manager.
- **Advanced utilities.** Provides access to some utilities that provide an added value to LinkEHR. They are mostly utilities to generate useful implementation resources such as implementation guides, schematron validation rules, or sample data instances.
- **Configuration.** Configuration options of LinkEHR Studio, that can be imported or exported to other installations.
- **Help.** Documentation of LinkEHR Studio and information about the tool and the license.

### 4.3 Toolbar

LinkEHR Studio toolbar provides a quick access to the most used functionalities of the tool. They are described in the next table.

<table>
<thead>
<tr>
<th>![Icon]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Icon]</td>
<td>Creates a new and empty archetype</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Creates a specialized archetype based on an existing one</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Open an archetype from the local file system</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Save the archetype being edited in the local file system</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Validate current archetype</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Access to the archetype edition view</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Access to the archetype mapping view</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Access to the transformation export view</td>
</tr>
<tr>
<td>![Icon]</td>
<td>In the archetype mapping view, this dropdown list allows selecting the legacy data source that is going to be mapped to the archetype.</td>
</tr>
<tr>
<td>![Icon]</td>
<td>Access to the ADL edition view</td>
</tr>
</tbody>
</table>
In the ADL edition view, allows searching a text inside the archetype

Generation of a mind map view of the archetype

Generation of a sample form for data entry based on the archetype

Access to the LinkEHR documentation

4.4 Archetype tree

The archetype tree is your main view of the archetype structure and contents. The root node of the tree shows the complete identifier of the archetype. Clicking on it provides access to the archetype metadata. In addition, from the archetype identifier you can navigate to the archetype definition structure to see the complete archetype contents (see Section 8.1).

The archetype tree contains two additional main nodes: language and ontology. Language allows translating the archetype textual descriptions into other languages (see Section 8.3). Ontology provides a unified access to the terminology bindings of the archetype (see Section 8.4).
4.4.1 Generic and specific archetype trees

The archetype tree can adopt two different forms. For all archetypes and reference models, the generic archetype tree shows the complete structure of the archetype, including all of its nodes. For some reference models (currently, ISO 13606, openEHR and HL7 CDA) an additional archetype tree is available. That new tree provides a simplified view of the archetype for those well-known reference models. This simplified view hides most of the complexities of the model and allows the person who is editing the archetype to be centered in the clinical structure that the archetype has to represent.
4.4.2 Archetype tree labelling options

The archetype tree-labels can be configured in three different formats, that change cyclically when the button is clicked: show only texts from archetype ontology (clinical view), show type name, node identifier and texts from archetype ontology (technical view), and show type name and node identifier (mixed view).
4.4.3 Filter current archetype

This control allows the user to define a filter that will be applied to the current archetype tree. This will only show nodes whose name is compliant with the filter. This is especially useful while checking for errors from the semantic validation or while mapping the archetype. A current filter can be removed pressing the ‘clear filter’ button.

4.4.4 Expand nodes

This button is used to expand or retract the archetype tree nodes. The use of the button is as follows: the first click on the button expands all tree nodes, while a second click returns the tree to the original expanded state (before pressing the button). More clicks after the second trigger the expansion and restore actions alternatively again.

Also, holding CTRL while pressing the button expands the tree one level more, and holding SHIFT retracts tree one level.
4.4.5 Filter archetype by its mapping

The purpose of this button (which is only available in the mapping view) is to filter current mapping archetype tree following several criteria like show only mapped nodes, nodes with object mappings, nodes mapped with constants or with any data source.

![Filter options](image)

4.4.6 Language selection

This control changes the language of the labels of the current archetype to the selected language. Only languages that have been configured in the Language section are available (see Section 8.3).

4.5 Constraints definition

The constraints definition area is the place to define the constraints and properties of the archetype. This area is a form that adapts to the selected type of node in the archetype tree. There are specific forms for the archetype metadata, for object nodes, for attribute nodes, for primitive nodes, for the archetype ontology, and the archetype language. You can find the details of how to define constraints in Section 8.
4.6 Additional information area

The following sections are available in the information area:

- **Node info**: This view provides information about the selected node in the archetype tree. This information includes the type of node (object node, attribute node, primitive node), its node id, the ADL path of the node, the equivalent ADL path in the reference model, the path of the node in a readable format, and the equivalent XPath of the node to be used in data instances.

- **Console**: Shows messages about the status of the tool. Mainly, when an archetype is loaded and it contains critical syntax errors, a parsing error is shown with informing about
the ADL line where the error was generated. The user can access to the ADL view (see Section 8.5.3) and try to correct the archetype manually.

- **Terminology**: Provides access to the internal terminology library of LinkEHR Studio. This library is not a substitute of a complete external terminology service, but provides access to the internal terminologies used by the ISO 13606, openEHR and HL7 CDA standards.

- **Local Repositories**: This view provides a list of the archetypes defined in the local repositories, classified by standard and reference model class. More local repositories can be defined in Configure LinkEHR ➔ Archetype repository (See 5.1.f)
5 LinkEHR Studio configuration

5.1 Configuration options

You can change the configuration of LinkEHR Studio in the menu Configuration→Configure LinkEHR. The dialog has several tabs:

a. **Application**: Configuration of the interface language of LinkEHR. Currently LinkEHR is only available in English and Spanish. You can also set the interval of time between autosave for the edited archetypes, and the information for connecting LinkEHR through a proxy server.

![Configure LinkEHR dialog](image)

b. **Visualization**: you can define the preferred visualization of the archetype tree labelling (to show the ontology description, classes’ names or both) and the default font for the archetype tree. You can also set to load the specific archetype editor by default when a known standard is used (see Section 4.4.1), to show the flat archetype view in specialized archetypes (see Section 7.3), and the alphabetical sorting of archetype nodes in the archetype tree. Finally, you can configure LinkEHR Studio in a colourblind palette.
c. **Paths**: you can change the default paths used by LinkEHR Studio, such as the location of the internal terminology XML files, the location of the reference models, and the location of the imported data source schemas.
d. **Default author**: You can configure the default information of the author of the archetypes in order to avoid retyping it every time you create a new archetype. The Default language refers to the primary language selected by default when creating a new archetype in the “Create new archetype” wizard.

![Configure LinkEHR](image)

- **Mapping**: Options related to the mapping edition
- **Simplify mappings.** When loading an existing mapped archetype, mappings are analyzed and converted to their simplest form.
- **Check semantic validation when passing to mapping view.** Tells LinkEHR if semantic validation should be performed when changing to mapping view. If the archetype is not valid, it will not be possible to enter into the mapping view.
- **Check for orphan mappings.** When a mapped archetype is mapped, some of the nodes may have been removed or relocated. This option tries to find new nodes where an existing mapping can be assigned.
- **Number of recent used functions.** Number of quick access mapping functions in the mapping definition window.
- **Autosave mapping.** Interval of time for autosave of mappings.
- **XQuery and XML syntax coloring.** Allows code coloring for the generated XQuery and XML tests.

f. **Archetype repository:** Configuration of archetype repositories where existing archetypes can be found. By default, all public openEHR CKM repositories are configured. Archetype repositories support the definition of local repositories, LinkEHR model manager repositories, and CKM-based repositories (both legacy API and Rest API)
g. **Terminology repository (beta):** Configuration of terminology services that can be queried. Currently this option supports the FHIR API terminology and proprietary Indizen terminology server – CliniTerm (formerly ITServer).
5.2 Exporting and importing configurations

All the configuration options, plus the complete set-up of the LinkEHR Studio environment, can be exported and then imported in a new LinkEHR installation. This includes the imported data sources, imported reference models, stored semantic patterns, and, most importantly, the active license of LinkEHR Studio. You can access the import and export options in the menu Configuration.

You can select which of the information will be exported. The result will be a *.LCFG file that can afterwards be imported in a different LinkEHR Studio installation.
6 Managers

Managers are the place to configure some of the basic LinkEHR Studio functionalities, including:

- The import and configuration of reference models used in LinkEHR Studio.
- The import and removal of semantic patterns.
- The import and removal of data sources.

You can access managers in the menu Managers.

6.1 Reference model manager

The Reference Model Manager is the place where you can import or remove a reference model in LinkEHR Studio. Reference models are the basis of archetypes, guiding the structure, class and attribute names that can be constrained in it. By default, LinkEHR Studio includes three reference models: ISO 13606, openEHR and HL7 CDA. However, it is possible to add any other model to the tool and use it as the basis for your archetypes.
6.1.1 Importing a new reference model

LinkEHR Studio supports the import of new reference models defined in XML Schema or the Basic Meta-Model (BMM) format of openEHR.

Open the reference model manager in the menu Managers → Reference Model Manager and click the Import RM button. Select one of the options:

- **From XSD Schema.** This is the standard XML Schema format for XML documents.
  - Introduce the organization name and reference model name (without blank spaces) that you want to use to identify the reference model.
  - Select the definition strategy of the XML Schema. Usually, the different elements of an XML Schema are defined as Complex Types `<xs:complexType>` tag).
  - The standard node identifier field lets the user to select which is the attribute of the reference model that will contain the link to the archetype or AT-code that describes the data. For example, in ISO 13606 this attribute is archetype_id and in openEHR it is archetype_node_id. In HL7 CDA the equivalent node would be template_id. This information is used during the mapping process, only when an archetype is used as a source schema.
  - Select the XML Schema file(s) that contain the reference model and press Next.
o LinkEHR Studio will analyze the Schema and generate a list of available entities. An entity is any class or type defined in the XML Schema that can be potentially a root of a new archetype. The list is filled with the entities in complexity order, as it is likely that the more complex entities will be the ones modeling structures. If a needed entity does not appear in the list you can move the “Entity complexity” slider in order to show less complex entities. They are hidden at first to simplify the whole process.

o Select the entities that will be the basis for creating new archetypes and click on the arrow to the right. This will show the selected entities in the “included” list. These are the root classes of new archetypes. For example, in ISO 13606 they are FOLDER, COMPOSITION, SECTION, ENTRY, CLUSTER and ELEMENT.

o Included entities will be the only ones available to create new archetypes and transformation scripts based on them.

```
Include reference model entities
Select the entities to be included. The entities are displayed in complexity order.

Net included
ENTITY_PART
XML
QUANTITY_RANGE
VLRQ
RTO
ENTITY_ROLE

Included
ENTITY_NAME

Entity complexity:
Less complexity More complexity
```

**ADDITIONAL INFORMATION**

XML Schema is a complex technology that allows many possibilities and variations when defining a schema. Although we have made our best effort to support most cases, the import process can fail under extreme circumstances while analyzing the XML Schema. In case you find any error, please contact us.
• **From RM file.** This is a proprietary LinkEHR format used to distribute new reference models by VeraTech. In case you receive a *.RM file, choose this option and select the file to import it directly into the tool.

• **From BMM File.** BMM (Basic Meta-Model) is a format defined by the openEHR Foundation. This format contains all the information needed to define a new reference model for LinkEHR.
  - Select the BMM files where the reference model is defined.
  - Introduce or correct the organization and the reference model name.
  - Select the entities that will be available for defining new archetypes and include them in the list at the right.
  - Click Generate to parse the BMM and generate the reference model specification.

![Select archetypable entities](image)

### 6.1.2 Reference model documentation

A documentation file is automatically generated for each reference model that is imported into LinkEHR Studio. This documentation describes several properties of the reference model that are useful for LinkEHR Studio, mainly related to visualization of the archetypes. The documentation file can be edited manually in the Reference Model Manager window. Select a reference model in the Available reference models’ list and the set of classes and attributes of that model will be loaded. For each class, its attributes are listed. In case of inheritance, attributes from parent classes are replicated in each of the child classes.

The following properties are global for each reference model:

• **Current language** selects the language used to document textual descriptions of the model. The documentation can be defined in multiple languages.
- **Node id path** defines the location in the reference model data instances where the archetype node identifier is stored. This usually corresponds to an attribute that is common to all classes. In case of working with XML data instances, this attribute can be an XML attribute. In that case, its name should start with the symbol `@`.

- **Model Class** is the class name and path that will be used to generate objects on the specific editor. For internal developers use only.

- **XSLT Path** is the path to the XSLT file that will be applied to transform XML data compliant with this reference model that is generated after a data transformation process.

The remaining options are specific for each node of the reference model. Select either a class or an attribute to edit its documented properties. Two field are especially relevant for the documentation of a reference model:

- **Short description** is used to display a tooltip associated to a menu item during archetype edition time, when a new class is added.
- *Long description* is displayed in the form associated to each type.

For classes, the following specific properties can be changed:

- **Is son of:** Choose which class is the parent of current class.
- **Is abstract:** Choose if current class is abstract.
- **Allows term constraint:** Indicates if this class represent information about coded values from a terminology, and thus can be changed by an archetype constraint binding (ac codes) in the archetype tree. See Section 8.2.5.3.
- **Form class:** Choose the java class that implements the form to edit this type of class. For developers internal use only.
- **Icon path:** The path to the icon of this kind of class. This icon will be shown in the archetype tree and the mind maps representations.
- **Navigation attribute:** The name of the attribute used to navigate to other main classes of the reference model (e.g. on a COMPOSITION class, **Navigation attribute** is “content”)
- **Allowed types:** Types that this generic class can handle (e.g. The **Allowed types** for IVL<T> are TS, PQ, DURATION...)

For attributes, the following specific properties can be changed:

- **Terminology:** Choose the default terminology used to populate the values stored in this attribute.
- **Visible:** Choose if an attribute is on the way from a reference model class to a leaf class (typically a data type). Used for specific reference model archetype trees and mind maps generation.
Allowed children: List of all classes that can hang from this attribute

Finally, you can add new nodes to the documentation manually by clicking the Add Node button. This is useful if you have created your reference model without using any definition file.

6.1.2.1 Mindmap generation

The following properties change the way mind maps are generated:

- **Navigation Attribute**: Probably the most important of all the modifications. Adding an attribute name that is part of that class (or a set of comma separated attribute names) tells LinkEHR which attribute or attributes should be used to navigate the archetype structure. For example, both ‘parts’ and ‘links’ are considered as navigable attributes of the ISO 13606 CLUSTER class. The archetype structure following these attributes will be also included in the mindmap that is generated. **Any reference model attribute not defined as a navigation attribute will not be included in the mindmap generation.** Inheritance is preserved by this property. This means that you can define a navigation attribute in a parent class and it will be added to all child classes.

- **Icon Path**: You can define the icon that would be attached to a given class in the mindmap. Most of LinkEHR internal icons (see Figure 4) can be used for defining the icon for a given object of the reference model. To use them, just write ‘icons/icon_name’ in the Icon Path field (changing icon_name to the corresponding icon you want to use). Absolute paths to any image in your disk can be also defined but be aware that no preprocess is made into the image, so sizes may vary depending of the source image.

6.2 Semantic pattern manager

This manager allows the configuration of semantic patterns in LinkEHR Studio. It allows adding a new semantic pattern from an existing archetype, remove it, or use current archetype being edited as a semantic pattern. You can find more information about semantic patterns, and about how to use them in Section 8.5.1.
Note that as semantic patterns are based on the archetype identifier, importing a second archetype with the same identifier will update current semantic pattern definition with the new one.

6.3 Data source manager

This manager is the place to import and configure data sources of LinkEHR Studio transformations. A data source is a definition of the structure of existing data, for example the XML Schema of a set of XML documents. Once a data source is imported, we will be able to map it to an archetype to, finally, generate a transformation program that will convert that existing data into data compliant with the archetype and the reference model used. To know more details about data mapping and transformation see Sections 9 and 10.

Data source manager shows a list of already imported data sources. You can add new data sources, remove existing ones, and import/export data sources to migrate their configuration to other LinkEHR installation.
6.3.1 Data source import

To import a data source means to import the original or legacy data schemas into LinkEHR Studio. Click Add button in the data source manager. The window that appears lets us to select the data source schema and to define several properties of it.

- **File**: The file that contains the definition of the data source schema. It can be one of the following formats:
  - XSD file: W3C standard XML Schema format. This is the recommended format to be used to define a data source.
  - RDB file: Deprecated. An old proprietary format of LinkEHR Integration module.
- MAD file: The current definition format of integration profiles in LinkEHR Integration module. This format defines an XML view over relational databases and can be directly used as a source for mappings in LinkEHR.
- Archetype: ADL archetype. You can map an archetype to another archetype and transform data between them.

You have to choose the format of the data source schema in the import window.

![Image of XML file selection]

### ADDITIONAL INFORMATION

LinkEHR transformation work over XML data. This means that relational databases are only supported as long as existing data is first extracted in XML format. LinkEHR Integration is a module of the LinkEHR Interoperability Platform that will help you with this task.

- **Alias**: The name that will be used internally to reference to the source when defining mappings. It should be a descriptive name.
- **Root**: XML tag that identifies the root of the source XML documents.
- **Id path**: By clicking the Browse button, we will get a summary of the imported schema. We must select the field or element where the main identifier of data is located. When talking about health data, this identifier should correspond to the patient identifier. This information is necessary to avoid merging data from two different patients accidentally. When the source is an archetype it is not needed to define this attribute.
- **External source**: XML Schema allows defining its contents in multiple files, by referencing internally their location. In those cases, importing the schema into LinkEHR may break these internal references. Checking this option will avoid to import physically the source XML Schema and just include a reference to its original location. Thus, the internal references will be respected and be still valid. If the schema files are removed from their original location, the source won’t be valid anymore.
7 Working with archetypes

7.1 Creating a new archetype

To create a new archetype, click the **New Archetype** button in the toolbar or the menu **Archetype** → **New**. You have to define the following parameters about the reference model you want to use as basis of your archetype, and about the archetype itself:

- **Organization**: organization that is responsible of the reference model. Usually they are Standardization Organizations, such as ISO, CEN, HL7 or CDISC, although it can be any other organization.
- **Reference model**: the reference model to be used. For example 13606 or CDA.
- **Entity**: the root entity or class of the archetype. Depending on the selected reference model, you will have here a list of the main classes that can be used as root of the archetype. The selection of one specific root type will depend on the requirements and the information you want to define in the archetype.
- **Concept**: the main name of the concept that the archetype is representing. This name will be part of the internal archetype identifier.
- **Language**: original language of the archetype. Although we can translate the archetype afterwards, we always use an original language to create it that we will use as the main reference for translations.
- **Use semantic pattern**: selection of a semantic pattern from those stored in the internal library. The list will only show those semantic patterns whose root entity is the same as the selected entity in this window.
### 7.2 Opening an existing archetype

Click the *Open* button in the toolbar or the menu Archetype→*Open* to select an ADL archetype and open it in LinkEHR Studio.

If the archetype is correct syntactically (ADL-conformant), you will see the archetype tree loaded with the archetype structure.

If the archetype is not valid (ADL with errors), an error message will be shown in the additional information area, indicating the line that generated the error. The archetype tree will not be built. You can edit the ADL code and try to correct it manually.

Click the ADL code is available and can be accessed through *Go to ADL* button. Looking at the ADL code will allow you to easily detect the error, fix it and recompile the ADL text (by clicking the *Go to ADL* button again) to generate the visual representation of the archetype.

### 7.3 Specializing an archetype

To create a specialization archetype, you only have to click the “Specialize Archetype” button of the toolbar or the menu Archetype→*Specialize*. A dialog will be displayed to select the parent archetype of the specialization and to introduce the specialization name.
Clicking “Ok” will create the specialization archetype, with the structure of the parent archetype already included. Then you can modify the specialized archetype the same way as described in step 7.

7.4 Saving an archetype

Both “Save” and “Save as...” allow the saving of the archetype in ADL form. “Save” overwrites currently selected archetype while “Save as...” allows to specify other places where to store the ADL file. For newly created archetypes, both “Save” and “Save as...” work the same way the first time using either option. Also, whenever a given archetype has been modified from the existing file and another option is selected in LinkEHR Studio that would replace it (open, specialize, import, etc.), a dialogue will appear informing that the contents have changed and you may want to save it.

7.5 Archetype import

LinkEHR supports a variety of formats as a basis to import an existing modelling effort into the tool. These include archetypes from external available sources and archetypes/templates defined in other formats.

7.5.1 Archetype Repository

This opens a dialogue to import archetypes from existing repositories, both online and local. Search type can be concept search (search in concept part), description search (search in concept and descriptions), or full search (search in any part of the archetype).¹

¹ Note: search capabilities depend on the actual API of the selected repository
New archetypes repositories can be added in LinkEHR configuration (see 5.1.f for additional details)

7.5.2 Open XML archetype

Opens an archetype that has been exported to XML (e.g. by the Export XML archetype described in 7.6.3)

7.5.3 Open Lkr file

LKR format is a compressed format that groups integration scenarios. This option opens a LKR file, opening the archetype and imports other available artefacts (source definitions, mapping definitions, rules, etc.)

7.5.4 Open OET template

OET is a de facto XML template definition format available. This format can be imported into LinkEHR Studio for template definition.

7.5.5 Open OPT template

OPT is the currently preferred template format for openEHR. This format can be imported into LinkEHR Studio for template definition.

7.5.6 Import FHIR

This import option allows importing a FHIR STU3 profile defined in json format as a FHIR archetype. After selecting the json file, a dialogue allows selecting further options.
Generate: differential/snapshot. This creates the archetype based on differential or snapshot sections of the FHIR profile. It is worth noticing that differential views should be preferred (as this goes along better with having an underlying reference model). However, snapshot generations can be useful for creating full mappings.

Concept: Concept name for the generated archetype.

Parse text constraints: Yes/No. Some FHIR STU3 profiles contain constraints in the form of vertical slash-separated strings (e.g. “a | b | c”). These constraints can automatically be translated into string constraints in the archetype.

Parse CDA mappings: Yes/No. Some FHIR STU3 profiles also provide mappings to HL7 CDA. This effectively generates a mapping definition that can be used to automatically generate a mapping from a HL7 CDA data instance to the given FHIR profile.

7.5.7 Import ADL2 archetype

This option allows importing ADL2 archetypes/templates as ADL1.4 archetypes. This uses archie library for parsing purposes.

7.6 Archetype export

LinkEHR allows exporting the archetype into different artifacts and reference materials. These go from the representation of the archetype in other definitions such as mindmaps, excel, etc. to other formats such as XML or Json schema.

7.6.1 Export mindmap

This option allows generating a mindmap file equivalent to the one shown inside linkEHR. If freemind is installed in the system, this export process also exports the archetype classes’ icons (Entry, Element, different data types) into freemind so they can also be viewed directly from freemind.
7.6.2 Export Excel

This option allows exporting an Excel file from the archetype structure and contents. Generated Excel contains both archetype description (metadata) and definition (structure and contents). In the definition, an archetype tree with their description, type, node identifier, constraint, terminology binding, occurrences, if the node is mandatory, and the archetype path are included.

![Excel Table]

7.6.3 Export XML archetype

This option exports current archetype in XML format. It can be opened by the import XML archetype option (7.5.2)

7.6.4 Export JSON Schema

This option generates a JSON Schema from the current archetype.

7.6.5 Export ADL for openEHR editor/Template designer

This option allows generating archetypes with the subset of ADL understood by openEHR editor or template designer.

7.6.6 Export Operational Template (OPT)

This option exports current archetype or template as an OPT.

7.6.7 Export Canonical Schema

This option exports a canonical XML Schema to ease the integration with providers unaware of archetypes. This also generates the transformation program to translate data in the canonical schema to the normalized one based on the archetype.
7.6.8 Export LKR

LKR format is a compressed format that groups all the artifacts generated by linkEHR (archetype, rules, mapping definition, etc.). This is used by other VeraTech products such as LinkEHR Model Manager.

7.7 Share archetype

Share archetype allows sending current archetype by email. It also allows to attach the mapping definition file, the source of the mapping (Schema or archetype) and even an XML data instance for testing purposes.

7.8 Archetype printing

Print archetype opens a print dialogue that generates and prints the corresponding archetype tree.
8 Edition of archetypes

8.1 Archetype metadata

Archetypes contain a set of metadata that can be modified to provide better attribution and allow better discoverability.

Archetype metadata contains: Lifecycle state, version, original language, copyright, keywords, purpose, use, misuse, and general details, and other details. If there is one metadata item that cannot be put into an existing field (such as license or original publisher), it can be put into general details or other details, as they are key-value pairs.

It also includes information about the original author, all the translators, and every other contributor to the archetype.

8.2 Constraint definition

Archetypes are created by constraining the reference model to narrow the constraints of your specific use case. Constraints are created by either adding nodes to the archetype tree or by narrowing the constraints on a selected archetype object or attribute.

8.2.1 Add and remove constraints

The basic add and remove constraint operations are made by right-clicking in the archetype tree. This provides a contextual menu that shows a set of allowed constraints to be created at a given
point in the archetype. Inside of object nodes, only the attributes allowed inside that kind of object are presented (and only if an attribute with the same name did not exist already). Inside of attributes, the set of valid type alternatives is presented (regardless if an object of the same type exists inside of the attribute or not). The contextual menu also provides the option to delete a given node of the archetype. Alternatively, the same contextual menu can be obtained by selecting a node and clicking on the “edit” option of the menu bar.

8.2.2 Object nodes

Object nodes allow the redefinition of occurrences, alongside the object text, description, comment, and/or terminology binding.

8.2.2.1 Occurrences

Occurrences define the number of times a given object can appear or repeat in under a given attribute.

Any number can be put in the minimum and maximum occurrences of an object, as long as a few rules are met: Minimum occurrences must be always equal or less than maximum occurrences, and both minimum and maximum occurrences must be compatible with the ones in the reference model or parent archetype. E.g. if parent occurrences are 1..4, it is not allowed to have a children occurrences of 0..* or 1..6. However, having occurrences of 1..1 is just fine.

Another rule is that the sum of all the minimum and maximum occurrences of all the objects in a given attribute must be compatible with the archetype cardinality. The unbounded check means that no upper limit has been defined for the occurrences.
8.2.3 Attribute nodes

Attribute nodes allow setting the existence (if a node is mandatory or not, or if the attribute is not allowed to appear in data). For multivaluated attributes, it is also possible to define the cardinality, alongside its uniqueness and if the objects inside the attribute are supposed to be ordered.

8.2.3.1 Existence
Both “obligatory” (mandatory) and “not allowed” checks allow the constraint of attribute existence. They must also be compatible with the parent archetype or reference model constraint. E.g. an optional attribute can be made obligatory, or an optional attribute made “not allowed”. However, a mandatory attribute cannot be made optional.

8.2.3.2 Cardinality
Similar to the occurrences, a pair of spinners containing minimum/maximum cardinality is presented. These represent how many objects could be potentially be put inside a given attribute. Unbounded means that no upper limit has been defined. These minimum/maximum cardinalities must be compatible with the occurrences of all the objects inside this attribute.

It is also worth noticing that even attributes with no cardinality (single attributes) may also contain more than one object, effectively creating alternatives. Usually, different cardinalities can be defined to simulate alternatives for multivaluated attributes.

In addition to that, there is an “ordered” check, which means that the objects inside must respect the definition order, and a “unique” check, which means that every object must be unique inside (non-repeated data).

8.2.4 Primitive types

Archetypes allow a set of primitive types to be added inside specific attributes. These special objects contain the constraints that primitive types must follow to be considered valid.

8.2.4.1 String type
For String types, both a constraint on the value and an assumed value (i.e. what value should be assumed if not provided) can be defined. String constraints can be either a default value (fixed value), a list of valid values, or a pattern. For the pattern, a library of well-known patterns is also included.
8.2.4.2 Integer type

Similar to the string type, integer type can also have both a constraint on the value and an assumed value (i.e. what value should be assumed if not provided). Integer constraints can be either a default value (fixed value), a list of valid values, or an interval of values.

8.2.4.3 Real (double) type

Double type can have both a constraint on the value and an assumed value (i.e. what value should be assumed if not provided). Double constraints can be either a default value (fixed value), a list of valid values, or an interval of values.
8.2.4.4 Boolean type
Boolean type constraints allow to specify which values are valid (both true and false, or true, or false) and their assumed value (true or false).

8.2.4.5 Date-time type
Date-time types allow to constraint a date-time pattern, a list of valid date-times, or a range of date-times as valid values. Also an assumed value can be defined.

8.2.5 Special nodes
In addition to object nodes and data types, archetypes also support other kinds of special objects to ease reuse of knowledge and data structures.

8.2.5.1 Internal references
References to other nodes inside the same archetype can be created with the “Create an Archetype Internal Ref” option. This allows you to choose the type of Internal reference, which in the end limits the available objects that can be referenced.
8.2.5.2 Archetype slots
Archetype slots provide means of referencing external archetypes. These included archetypes are then logically included in the archetypes, and usually are expanded when defining templates or generating mappings. Typically, allowed archetypes are expressed as a regex to allow not only specific archetypes but also specialized archetypes from them. The archetype slot dialogue provides a mini editor to ease the correct definition of these regexes.

8.2.5.3 Constraint references
These special nodes allow defining a constraint-based on an external service, such as a query to a terminology service
8.2.5.4 Domain types

A domain type is a special type of constraint to support specific syntaxes of particular reference models. Domain types always have an equivalent standard ADL equivalence.

LinkEHR supports OpenEHR domain types such as DvOrdinal or DvQuantity. These constraints should be transformed when defining mappings and other processes that involve data types (although the tool does this automatically for most of the processes that do not modify the original archetype).

8.3 Archetype language translations

Archetypes can be translated into different languages. In principle, every translation should be made by using the original language texts as a basis. Archetypes are capable of storing the name and credentials of the person that made a given translation.
8.3.1 Language management

In the archetype tree, there is a “Language” item, which allows managing archetype languages. Creating a new translation is as simple as pressing the “New language” button and filling all the metadata about the translation in a given language. This adds a new language to the tree.

8.3.2 Translating an archetype

Every node text in a new language translation is commonly represented as a new text created by adding an asterisk to the original language text. This allows the quick review for missing translation strings and to help with the meaning of the existing node. Translations should be made by official translators or at least by domain-aware personnel.

8.4 Archetype ontology definition

Archetype ontology (also called ‘vocabulary’ in newer versions of ADL) is the section where all the semantics are attached to the archetype tree structure. This ontology definition encapsulates the strings for each one of the labels and descriptions of the tree nodes and constraints and the terminology binding of both labels and constraints. This ontological information can be defined in each one of the object nodes in the tree, but also as a review in the ‘Ontology’ node of the archetype tree.
8.4.1 Term definition

Term definition encapsulates all the objects labels and descriptions for each language (both original language and all its translations) for every object node in the archetype (atXXXX codes). In addition to text and description, additional keys (such as ‘comment’) can be added to a given object in the archetype tree.

8.4.2 Constraint definition

Similar to term definitions, constraint definition encapsulates all labels and descriptions for the constraint reference nodes (acXXXX in ADL1.4)

8.4.3 Term binding

To give semantics to the object nodes in the archetype, it is also possible to create a term binding. In a term binding, a code in a given terminology/ontology is selected as the one that represents best the meaning of that node. For term bindings, only one code per terminology should be selected. However, it is allowed to have bindings to more than one terminology.

8.4.4 Constraint binding

When creating string constraints, it’s possible to define the set of valid string values that apply to a given point in the archetype. However, this approach has a severe limitation: only the strings included are valid ones, so no translations are allowed. To solve this limitation, a constraint definition can be created instead for each one of the string alternatives: with this, the value stored in data is the reference to the code in the vocabulary, which allows for better visualization and querying.
8.5 Advanced edition

8.5.1 Semantic patterns

LinkEHR can also define and use semantic patterns. Even if the concept of semantic patterns is more powerful (in theory you can define a complete archetype just by selecting which semantic patterns it uses), the current approach does not support the application of two patterns over the same structure. We call semantic patterns to the reusable structures that can be included in any part of the archetype where they are semantically valid. Note that semantic patterns are not limited to the valid entities of the reference model, but also to any other entity on the reference model (such as data types or context classes).

Managing Semantic patterns

LinkEHR has a manager to manage specifically the semantic patterns. It can be accessed under the x menu. This is a capture of the manager.

This manager is quite simple. The three options available are to add an existing archetype as a semantic pattern, to delete one of the current available semantic patterns of the list, and to use the current opened archetype as a semantic pattern. Note that as semantic patterns are based on the archetype identifier, importing a second archetype with the same identifier will update current semantic pattern definition with the new one.

Transform an archetype node into a semantic pattern

Another way to add a new semantic pattern into LinkEHR is to right-click an object from the archetype tree and then click on “To Semantic Pattern”. This will take the sub-branch from the archetype and transform it into a semantic pattern.
One thing to take into account is that semantic patterns are self-contained, so the existence of internal references to places of the archetype outside the reach of the new concept is not allowed as it would create invalid semantic patterns (having references to a part inside of the pattern is Ok). You can use the operations over the archetype internal references that linkEHR support to solve this before the generation of the semantic pattern.

**Inserting a Semantic Pattern on the Archetype**

Whenever is semantically valid to add a semantic pattern linkEHR will add the option “Insert a semantic pattern” to the right click menu.

Once this pattern is selected it will be included into the archetype with all the structures and ontological information available on the semantic pattern (at codes will be updated to fit into current archetype).

As a final note, you can also use a semantic pattern for copy & paste operations (although that can be accomplished faster with drag & drop functionality currently implemented in linkEHR)
8.5.2 Edition of archetypes without reference model

When the selected reference model is not recognized as an imported model into LinkEHR, you can still edit the archetypes in an unconstrained way. For each object, any number of attributes can be created. For each attribute, you must select if it’s a single or multivalued attribute.

Similarly, same thing applies for the object inside the attributes, where an object, an archetype slot, an internal reference or a specific data type can be selected.

Once the node has been created, their constraints, text, terminology bindings, etc. can be created in the normal way.

8.5.3 ADL edition

Explaining the ADL language is beyond the scope of this document, but you can learn about it in WWW.

8.6 Archetype validation

The “Validate archetype” button checks the archetype for any incongruence of its constraints with the reference model (or parent archetype, if available). Even if LinkEHR studio only allows you to create valid constraints in the archetype, some errors can involve more than one node (e.g. compatibility between attribute cardinality and children occurrences) so they must be validated on their process. LinkEHR also offers the option to require that the archetype is valid when going to mapping view. This option can be enabled or disabled in LinkEHR configuration.

8.7 Rules definition

Despite being able to define constraints into the archetype is a powerful mechanism to ensure data quality, there are times where more powerful constraints are needed. These advanced rules usually involve more than one path in the archetype or depend on external operations.
8.7.1 Variables

For variables, a name must be provided, alongside either a path or an expression. The path includes all paths in the archetype. Expressions are more powerful, as allow the reuse of variables into more complicated paths, such as external terminology or other available REST services. For the expression, a new function was added `@validateCode($subset_uri,$code)`, that allows testing if a given code is contained into the subset defined by the subset_uri.

8.7.2 Rules

Rules combine one or more variables to provide a Boolean value for validation purposes (or any value in case of generation). Every rule needs a name/explanation of the rule and an expression that can be evaluated. Adding or editing a rule shows an editor similar to the mapping editor (described in section 9.4). The main difference is that not all functions available in mapping are available in rules definition, as not all functions are supported for rules.

8.8 Advanced utilities

LinkEHR Studio provides a set of advanced utilities for different scenarios, such as validation, generating implementation guides or working with different standards.
8.8.1 Export XML instance with options

This option allows the generation of a set of generated sample instances that can be used for load tests. A dialogue will be shown containing the following options:

- **Output directory**: Folder where XML data instances will be created
- **Instance number**: Number of instances to create
- **Prefix**: Prefix to put on generated XML files
- **Extension**: Extension of the generated file
- **Generate optional elements**: Should optional objects be generated?
- **Generate optional attributes**: Should optional attributes be generated?

![XML instance options dialog](image)
Complete with reference model: Should archetype be completed with the underlying reference model?
Generate comments: Should comments on the XML elements be generated?
Generate ADL: Should ADL file be output in the folder as well?
Generate summary HTML: Should an HTML with links to the XML instances be generated?
Value generation:
  - Limit random numeric values to: Select a max value for the numerical random values
  - String value generation: Select one type of string value generation
    - Empty: No value
    - Node label with default string: attribute name + “_value”
    - Nearest atxxxx value: Puts nearest object text value
    - Nearest atxxxx or attribute name: Puts nearest object text value or attribute name
    - Node label with custom string: attribute name + custom string
    - Regex: Generates strings with given regexes2.
  - Generate errors in instances: Process tries to generate values that will not follow archetype constraints. This process depends does not work if no constraint has been defined at all (e.g. any string is allowed or every integer is valid)
  - A preferred number of repetitions: Number of times an object with multiple occurrences could be repeated.
  - On choice: Select either ‘choose first’ or ‘choose random’ (by default, choose randomly)

There are two alternatives for this process, the newer one is faster, and for the most part completely equivalent.

8.8.2 Schematron

Schematron is a language used for the validation of XML data instances. LinkEHR Studio can generate the Schematrons rules used to ensure a data instance follow the constraints in the archetype. When selecting the option, the following dialog will appear

2 Be careful with “.*” regexes, as they may generate any UTF-8 character. This can be useful for testing edge cases, but maybe is not what you are looking for
This first dialog allows selecting the following options:

- Exclude attributes: A comma separated string containing the attribute names to be ignored when generating the schematron rules.
- The set of rules to be generated: Three different options are presented, consistency rules (set of rules for general validation purposes), completeness rules (set of rules used for calculating how complete is a data instance), and specific rules, which shows a list of all available rules to select which ones are needed in the analysis.

- Generate class rules: These are rules that will come from the reference model types, such as if a given attribute is valid for a given type or a given attribute is from a valid type.

When a set of rules has been selected and the “Generate” button is pressed, the following dialogue will appear.
In this dialog, the schematron rules are shown. “Test schematron” allows the user to select an XML file to see how many rules execute and how many pass/fail. “Save schematron” saves this rules file to be executed later on. Additional rules defined in the Rules part of the archetype tree will also be included into their own section of schematron rules. Incorrect custom rules or variables will be removed.

It is also worth noticing that the contents of the dialog can be changed. This can be useful for adjusting some rules and testing them before saving. That is the reason because there is a “Regenerate Schematron” button, as it generates schematron again from the options selected in the first dialog.

**ADDITIONAL INFORMATION**

The difference in this functionality between free LinkEHR Studio and the licensed version is that in the free version schematron cannot be saved.

8.8.3 Implementation guide

This option allows for the generation of multilingual implementation guides from current archetype. First, you must select a directory where the implementation guide will be generated. After that, a dialog will be shown containing the following options:

- Generate a table of contents: Check to generate a Table of Contents (ToC) with links to every other section in the guide
- Generate mindmap: Check to generate an interactive mindmap to be included in the HTML implementation guides.
- Generate metadata: Check to include archetype metadata on the output implementation guide.
- Guide format: Choose either Word or HTML. Some options (such as the interactive mindmap or the form) will not be available for both formats.
- Generate NRL rules: Check to generate archetype constraints in NRL rules language. These rules styles are available in both English and Spanish.
- Generate rules in HL7 style: Check to generate archetype constraints in a readable HL7 style (SHOULD, MUST, etc.).
- Generate XML instances: Check to attach a XML instance for each object in the archetype.
- Generate schematron: Check to attach schematron code to the implementation guide.
- Generate value set tables: Check to attach a section with all valueset tables.
- Generate sample form: Check to attach a section with a sample form generated from archetype.
- Generate Reference Model table: Check to add a section with Reference Model information.
- Generate ADL attachment: Check to attach a section with ADL source code.

**ADDITIONAL INFORMATION**

The difference in this functionality between free LinkEHR Studio and the licensed version is that in the free version only first 10 classes will be generated.

---

³ http://nrl.sourceforge.net/
8.8.4 Implementation guide from directory

Same as before, but the process is applied over a set of archetypes in a given directory. This is the recommended way of generating implementation guides if more than a document model exists or slots exist in current archetype.

8.8.5 Export HTML report

Select a directory and obtain a report with stats of the included archetypes. This includes information such as a number of different classes, where each archetype is included, etc.

8.8.6 Parse XML instance

This option allows generating an archetype from a given XML instance in a known reference model (such as HL7 CDA). This will try to infer the constraints from the values contained in the data instance

8.8.7 Generate FHIR DSTU profile

This option generates a legacy DSTU profile from current (FHIR) archetype. This kind of profile definition is no longer common in FHIR world.

8.8.8 Generate Natural language rules

This option outputs archetype constraints as NRL rules.

8.8.9 Extract entities as slots

Analyzes the archetype and extracts all available root reference model entities as individual archetypes and references them as slots.

8.8.10 Transform openEHR to ISO13606

This option generates an equivalent ISO13606 archetype from a given openEHR one. This process also generates all ISO13606-part 3 meaning codes to correctly identify source classes

8.8.11 FHIR

This submenu groups several FHIR related transformations.
8.8.11.1 Transform FHIR to openEHR
This option opens a dialog to transform a given FHIR Resource into equivalent openEHR classes.

It has the following options:

- Select base openEHR class: Choose either CLUSTER or GENERIC_ENTRY as the base openEHR class.
- Generate: Choose either snapshot or differential parts of the resource to generate the archetype from.
- Source resource: Path to the resource.
- Target folder: Path to the output folder.

8.8.11.2 Transform openEHR to FHIR Observation
This option opens a wizard to transform a given openEHR archetype to a FHIR Observation Profile. The archetype is analyzed to provide a tentative grouping, which can be changed by the user. This transformation process can also generate Composition and Bundle profiles (if desired), and also ValueSet profiles from the archetype internal subsets. Objects can also be defined as Extensions and will be created like that. The wizard contains the following parameters:

- Output path: Folder where all the generated profiles will be generated
- Path table: This table contains all candidate paths to be included in the observation. Checks can be used to avoid the generation of some paths if needed. It also allows the user to select the target type of a node (i.e. a given object can be selected as the value part of an Observation resource, a component part, or even as defined in an Extension). By default type is a component. This table also contains the grouping, which means that every component sharing a group will be included in the same Observation Profile.
- Base uri: Base uri used for the identification and referencing of profiles, value sets, and extensions
- Add translations: Adds translations to the generated ValueSet or not.
● Create empty FHIR archetype: Creates an empty FHIR archetype for mapping purposes.
● Generate Bundle profile: Check to generate a Bundle profile containing all other generated resources.
● Generate Composition profile: Check to generate a Composition profile containing all generated Observations.

8.8.11.3 Transform openEHR to FHIR Questionnaire
This dialog allows transforming an openEHR archetype into the equivalent FHIR Questionnaire instance. This transforms archetype structures and types into questionnaire items and answers. It also generates the ValueSets needed in these answers.

Dialog contains the following options:

● Output directory: Folder where the questionnaire and value sets will be generated
● Base uri: Base uri used for the identification and referencing of questionnaire and valuesets
● Transform DV_TEXT to: Choose if DV_TEXT data types would be translated to text or string.
8.8.11.4 Transform archetype to FHIR Logical Model
This option transforms current archetype into a FHIR Logical Model. Dialog only accepts output path as a parameter.
9 Mapping of archetypes

9.1 Transformation archetypes

Archetypes are formal definitions of domain concepts based on a particular reference or data model. When we are working with legacy and non-standardized data, archetypes can be considered as a view that provides abstraction in interfacing between data sources and reference models used to communicate clinical data.

Since health data to be made public resides in the underlying data sources, it is necessary to define some kind of mapping information that links entities described in the archetype (object nodes and attributes) to data elements in data repositories (e.g. elements and attributes in the case of XML documents, tables and attributes in the case of relational data sources). An integration archetype is considered to be a mechanism that binds an archetype definition with existing data schemas. Therefore:

**Integration Archetype = archetype + mapping specification**

Imagine a very simple reference model for vehicles which states that a vehicle can have one to many wheels and transport one to many passengers (Figure 31).

![Figure 31. Vehicle reference Model](image)

We can define a new archetype, for example for representing a bicycle, by constraining the reference model. In this case, we define the concept bicycle as the vehicle with only two wheels (Figure 32).
Finally, if what we want is to generate instances of bicycles, it is necessary to merge the constraints of the archetype with all the information that has not been constrained of the reference model (Figure 33). This is what we call comprehensive archetype and it is the basis or target schema that will be mapped to the original data source schemas, since we do not only want to standardize the archetype related data but also all the surrounding data, for example, the context information supported by the reference model classes.

Comprehensive archetypes are automatically generated by LinkEHR when entering into the mapping view of the editor and substitute the archetype tree of the archetype edition view.
9.2 Mapping of archetypes to data sources

In our context the mapping specifications must define how to create archetype instances from the content of the data sources. It should be noted that every valid instance of an archetype is also instance of the reference model. Therefore, if the reference model is a standard, such as ISO 13606, the generated instances are compliant with that standard. Target archetypes may be created ad-hoc to mimic the legacy data or it may preexist, e.g. when it is drawn from a public archetype repository.

Archetypes are used to model arbitrary complex domain concept without any consideration regarding the potential internal architecture or database design of EHR systems. As a consequence, complex and expressive mapping specifications are necessary due to the low similarity between archetypes and EHR system. We have tried to incorporate to LinkEHR expressive mappings that not only specify value couplings but also structural mappings.

There exist two kinds of mapping specifications: atomic attribute mappings and object mapping. Atomic attribute mappings define how to obtain a value for an atomic attribute of an archetype (C_PRIMITIVE_OBJECTs, i.e. attributes that contain simple data types such as strings, integers, booleans…) by using a set of values from the data sources. For this purpose transformation functions and conversion, tables can be used. The simplest kind of attribute mapping is the identity function which copies a constant or a single source value into a target value.

LinkEHR requires the definition of an attribute mapping at least for each mandatory atomic attribute of the archetype. For each constrained class there exists an object mapping which contains both the query to be used to retrieve all the data necessary for generating data instances and the set of attributes that identify univocally the instances of the class. The combination of both components allows the conversion from source data to XML documents compliant with the Reference Model.

Archetype designers are responsible of defining the atomic attribute mapping and the system tries to generate automatically from them a set of candidate object mappings by taking into account the structure of the Reference Model entity, the constraints defined in the archetype and the integrity constraints of data sources. This approach alleviates the work of defining how to populate archetypes since it is easier for the designer to indicate which data elements of the data sources are relevant to a certain archetype attribute, rather than to specify the possible
complex query required to extract and transform all the relevant information, which may involve many data structures possibly from several data sources.

Once all the mandatory attribute mappings are defined, LinkEHR can generate an XQuery script that will transform legacy XML data instances into standardized XML instances following the archetype definition and reference model structure.

**NOTE**: The generated XQuery script is directly applied to XML data. A mechanism to permit this application to relational data sources transparently for the user is being developed.

### 9.3 Archetype mapping interface

#### 9.3.1 Archetype mapping tree

After opening an existing archetype or creating a new one (please, see the archetype edition manual also available at the web page), to perform the mappings we must enter into the *mapping view* by clicking the button of *Map View* (Figure 34):
At the left side we can see the comprehensive archetype tree that has been automatically generated. Colours indicate the state of mappings:

- **Black**: This node does not need to be mapped because it or its son nodes are not mandatory, but the node can be mapped anyway if you wish.
- **Red**: This node is mandatory (including the full archetype path to reach it) and has not a defined mapping or a single value in the archetype definition.
- **Green**: This node is mandatory and has already been mapped.

If we click a leaf or atomic node (an integer, string, double, Boolean or date/time) the right side of the editor changes to show the mapping interface for that node (Figure 35).
NOTE: If we want to map a node that inside an object defined as an Archetype internal reference, we can right-click that node and select the option Unfold. We can also double-click the internal reference node. This will create a clone of the referenced node to allow mapping it. The original archetype definition will not be affected by this action.

9.3.2 Archetype mapping table

Mapping view is divided in two main parts:
First, there is a field to introduce a comment about the current mapping. Second, the section *Transformation functions* contains the table of mappings and transformations that are applied to get a value for the archetype attribute. This table has two columns, *Condition* and *Mapping Function*. The first one permit to specify filtering predicates which describe the conditions that source data must satisfy to be used in a transformation function. Note that the order is relevant and only the first applicable function is used. The different rows of the table are evaluated consecutively, that is, the first-row condition is evaluated first, the second condition is evaluated if first fails, and so on. It is recommended to add a ‘default’ *true* condition as the last one to assure that we get a result. If a condition is accomplished, then the transformation function is applied. Both conditions and functions use the same syntax and will be explained in the following sections of this document.

### 9.4 Mapping definition

An attribute mapping is defined as a set of pairs (filter, function). The filter defines the conditions that source data must satisfy to use the corresponding function. When a filter is missing it is supposed that its value is true; therefore it will be applied always. LinkEHR allows the definition of multiple filter-function pairs in the specification of a single value mapping. For instance, the following table contains a very simple value mapping transforming gender codes. Note that the order is relevant and only the first applicable function is used, consequently the last filter acts as a ‘default’ condition.

<table>
<thead>
<tr>
<th>Filter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/Patient/gender=&quot;M&quot;</code></td>
<td>0</td>
</tr>
</tbody>
</table>
Conditions and mapping functions that are defined at the transformation table use the same syntax. In LinkEHR a wide range of functions are available, apart from the typical arithmetic, boolean and string functions, it also supports time and date functions and archetype related functions. For instance, it includes functions for the transformation of source time and date values into values conforming to the international standard ISO 8601 for date and time representation. Archetype related functions allow retrieving archetype metadata such as entity identification and linguistic entities defined in the ontology section like the text and description attached to archetype entities. By double-clicking the appropriate field of the table, the expressions editor will appear (Figures 38, 39, 40, 41, 42 and 43).

The main text field is where the expression is edited. It can be validated syntactically by clicking the button Validate. The different tabs of the window give access to the available operators and functions.

**NOTE:** String literals must be always double quoted. Parenthesis can be used anywhere in the expression. Equality operator is =. References to relational data source elements always begin with the symbol $. References to XML data sources are expressed on XML paths

### 9.4.1 Data source selection

This dropdown allows the selection of the source to be used in mapping specifications. If a source changes when mappings have already been defined, older mappings will be lost if they do not exist in the newly selected source

### 9.4.2 Mapping expression syntax

Mappings can be composed by the use of operators, functions, paths to the source, and constants. Every operation and function has an output type and a set of typed parameters (which can be empty). Mapping expression syntax ensures that used types are compatible in the place where they are used. In addition to that, the output type of the mapping function must be compatible with the archetype node where the function is applied.
9.4.3 Common operators

LinkEHR mapping supports a wide range of arithmetic and logical operators.

Arithmetic operators include:

- $A + B$ for adding.
- $A - B$ for subtracting.
- $A \times B$ for multiplication.
- $A \div B$ for dividing.
- $A \% B$ for the module operation.
- $A \text{ IDIV } B$ for the round down division.

Logic operators include:

- $A < B$ for expressing ‘$A$ less than $B$’.
- $A > B$ for expressing ‘$A$ more than $B$’.
- $A \leq B$ for expressing ‘$A$ less than or equal to $B$’.
- $A \geq B$ for expressing ‘$A$ more than or equal to $B$’.
- $A = B$ for expressing ‘$A$ equals $B$’.
- $A \neq B$ for expressing ‘$A$ not equals $B$’.
- NOT for the negation of the subsequent expression (can use parenthesis).
- OR for expressing that either of the expressions could be true (can use parenthesis).
- AND for expressing that both expressions should be true (can use parenthesis).
- TRUE for a true predicate.
- FALSE for a false predicate.

Typically an operation can be applied if the types of both are compatible. E.g. you can add two numbers, two float numbers, or even a date and duration. Note that two strings cannot be concatenated with the add operator and the concat operator should be used instead.

In addition to these operators, a general function for null checking is also provided.

<table>
<thead>
<tr>
<th>nullCheck</th>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>@isNull</td>
<td>&lt;Bool&gt; @isNull(&lt;Path&gt;)</td>
<td>Returns a Boolean value depending on if the source path is null or not.</td>
<td>@isNull(patient/deathdate) ⊆ false</td>
</tr>
</tbody>
</table>
### 9.4.4 Math functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td><code>&lt;Num&gt; @abs(&lt;Num&gt;)</code></td>
<td>This function gets an integer or real number and returns its absolute value as an untyped number.</td>
<td>@abs(3.5) ➞ 3.5</td>
</tr>
<tr>
<td>ceiling</td>
<td><code>&lt;Num&gt; @ceiling(&lt;Num&gt;)</code></td>
<td>This function gets an integer or real number and returns the smallest integer that is not less than the number argument as an Integer.</td>
<td>@ceiling(5) ➞ 5</td>
</tr>
<tr>
<td>floor</td>
<td><code>&lt;Num&gt; @floor(&lt;Num&gt;)</code></td>
<td>This function gets an integer or real number and returns the largest integer that is not greater than the number argument as an Integer.</td>
<td>@floor(5) ➞ 5</td>
</tr>
<tr>
<td>round</td>
<td><code>&lt;Num&gt; @round(&lt;Num&gt;)</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Description
Rounds the number argument to the nearest integer.

### Examples
- `@round(5.1)` 🡪 5
- `@round(5.5)` 🡪 6
- `@round(-5.5)` 🡪 -5

---

### round-half-to-even

**Use**
<Num> @round-half-to-even(<Float>,<Long>)

**Description**
Rounds a real number with the required decimals.

**Examples**
- `@round-half-to-even(345.678,1)` 🡪 345.7

---

### toFloat

**Use**
<Float> @toFloat(<Num or String>)

**Description**
Transforms a Number or String to a Float.

**Examples**
- `@toFloat("26.1")` 🡪 26.1

---

### toInteger

**Use**
<Int> @toInteger(<Num or String>)

**Description**
Transforms a Number or String to a Integer

**Examples**
- `@toInteger("25")` 🡪 25

---

### random

**Use**
<Num> @random()

**Description**
This function introduces a random number.

**Examples**
- `@random()` 🡪 4
### 9.4.5 String functions

<table>
<thead>
<tr>
<th><strong>left</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
<td><code>&lt;String&gt; @left (&lt;String&gt;,&lt;Int&gt;)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Trims a string from the left a number of characters.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td><code>@left(&quot;abcd&quot;,1) ☑ “a”</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>right</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
<td><code>&lt;String&gt; @right (&lt;String&gt;,&lt;Int&gt;)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Trims a string from the right a number of characters.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td><code>@right(&quot;abcd&quot;,1) ☑ “d”</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>starts-with</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
<td><code>&lt;Bool&gt; @starts-with (&lt;String&gt;,&lt;String&gt;)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Returns if a string starts with another string.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td><code>@starts-with(&quot;abcd&quot;,&quot;abc&quot;) ☑ true</code>&lt;br&gt;<code>@starts-with(&quot;abcd&quot;,&quot;cda&quot;) ☒ false</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ends-with</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
<td><code>&lt;Bool&gt; @ends-with (&lt;String&gt;,&lt;String&gt;)</code></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Returns if a string ends with another string.</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td><code>@ends-with(&quot;abcd&quot;,&quot;fg&quot;) ☒ false</code>&lt;br&gt;<code>@ends-with(&quot;abcd&quot;,&quot;cd&quot;) ☑ true</code></td>
</tr>
</tbody>
</table>

<p>| <strong>lower-case</strong> |  |</p>
<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;String&gt;@lower-case(&lt;String&gt;)</code></td>
<td>Returns the lowercase string of the string.</td>
<td><code>@lower-case(“CamelCase”) ↩ “cammelcase”</code></td>
</tr>
</tbody>
</table>

**upper-case**

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;String&gt;@upper-case(&lt;String&gt;)</code></td>
<td>Returns the uppercase string of the string.</td>
<td><code>@upper-case(“CamelCase”) ↩ “CAMELCase”</code></td>
</tr>
</tbody>
</table>

**capitalize-first**

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;String&gt;@capitalize-first(&lt;String&gt;)</code></td>
<td>Returns the string with the first letter in uppercase.</td>
<td><code>@capitalize-first(“a simple text”) ↩ “A simple text”</code></td>
</tr>
</tbody>
</table>

**compare**

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Num&gt; @compare(&lt;String&gt;)</code></td>
<td>Returns an integer (-1, 0, 1) resulting of the comparison of the strings.</td>
<td><code>@compare(&quot;a&quot;,&quot;b&quot;) ↩ -1</code>&lt;br&gt;<code>@compare(&quot;a&quot;,&quot;B&quot;) ↩ 1</code>&lt;br&gt;<code>@compare(&quot;ab&quot;,&quot;abc&quot;) ↩ -1</code>&lt;br&gt;<code>@compare(&quot;abc&quot;,&quot;abc&quot;) ↩ 0</code></td>
</tr>
</tbody>
</table>

**contains**

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Bool&gt; @contains(&lt;String&gt;, &lt;String&gt;)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Returns if a string contains another string.</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td>@contains(&quot;ab&quot;,&quot;abc&quot;)=false @contains(&quot;abc&quot;,&quot;ab&quot;)=true</td>
<td></td>
</tr>
</tbody>
</table>

### matches

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Bool&gt; @matches(&lt;String&gt;, &lt;Regular Expression&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns if a string matches a regular expression.</td>
</tr>
<tr>
<td>Examples</td>
<td>@matches(&quot;abc&quot;,&quot;a.*&quot;)=true</td>
</tr>
</tbody>
</table>

### concat

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;String&gt; @concat(&lt;String&gt;, &lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Concatenates two strings (resulting in another string). In order to ease the use of multiple concat operations in an expression the shortcut syntax <code>+++</code> can be used in mapping definition</td>
</tr>
<tr>
<td>Examples</td>
<td>@concat(&quot;abc&quot;,&quot;def&quot;)=&quot;abcdef&quot; &quot;a&quot;+++&quot;b&quot;+++&quot;c&quot; @concat(&quot;a&quot;,@concat( &quot;b&quot; , &quot;c&quot;))=&quot;abc&quot;</td>
</tr>
</tbody>
</table>

### index-of-string-first

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @index-of-string-first(&lt;String&gt;, &lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns an integer representing the first position of a substring within a string.</td>
</tr>
<tr>
<td>Examples</td>
<td>@index-of-string-first(&quot;abc&quot;,&quot;b&quot;)=1 @index-of-string-first(&quot;abc&quot;,&quot;z&quot;)=-1</td>
</tr>
</tbody>
</table>

### normalize-space

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;String&gt; @normalize-space (&lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the string without all the spaces and indentation characters.</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Examples</td>
<td>@normalize-space(&quot;   abc   &quot;)=&quot;abc&quot;</td>
</tr>
</tbody>
</table>

### substring

#### Use

<String> @substring(<String>, <Num>)

#### Description

Returns the string that is the substring of the string from a position.

#### Examples

@substring("abcd",2)="bcd"

### substring

#### Use

<String> @substring(<String>, <Num>,<Num>)

#### Description

Returns the string that is the substring of the string from a position with a length given.

#### Examples

@substring("abcd",2,2)="bc"

### substring-before

#### Use

<String> @substring-before (<String>, <Num>)

#### Description

Returns the string that is the substring that appears before the first occurrence of another specified string.

#### Examples

@substring-before("abcd","b")="a"

### substring-after

#### Use

<String> @substring-after(<String>, <Num>, <Num>)

#### Description

Returns the string that is the substring that appears after the first occurrence of another specified string.

#### Examples

@substring-after("abcd","b")="cd"
<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td><code>&lt;String&gt; @replace(&lt;String&gt;, &lt;String&gt;, &lt;String&gt;)</code></td>
<td>Replaces all the occurrences of the regular expression by the chosen string.</td>
<td><code>@replace(&quot;baba&quot;,&quot;b&quot;,&quot;cd&quot;)=&quot;cdacda&quot;</code></td>
</tr>
<tr>
<td>replace-first</td>
<td><code>&lt;String&gt; @replace-first(&lt;String&gt;, &lt;String&gt;, &lt;String&gt;)</code></td>
<td>Replaces first occurrence of the regular expression by the chosen string.</td>
<td><code>@replace-first(&quot;baba&quot;,&quot;b&quot;,&quot;cd&quot;)=&quot;cdaba&quot;</code></td>
</tr>
<tr>
<td>string-length</td>
<td><code>&lt;Num&gt; @string-length(&lt;String&gt;)</code></td>
<td>Returns the length of the string.</td>
<td><code>@string-length(&quot;22&quot;) = 2</code></td>
</tr>
<tr>
<td>toString</td>
<td><code>&lt;String&gt; @toString(&lt;Value&gt;)</code></td>
<td>Transforms parameter to a string.</td>
<td><code>@toString(22) = &quot;22&quot;</code></td>
</tr>
<tr>
<td>base64-encode</td>
<td><code>&lt;String&gt; @base64-encode(&lt;Value&gt;)</code></td>
<td>Transforms anything to a base64 string.</td>
<td><code>@base64-encode(&quot;acde&quot;) = &quot;YWNkZQ==&quot;</code></td>
</tr>
</tbody>
</table>
### 9.4.6 Time functions

#### toISODate

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Date&gt; @toISODate(&lt;String&gt;,&quot;yyyy-MM-dd&quot;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Transforms a string to an ISO date, being first parameter the date you want to read and the second parameter the pattern to read it</td>
</tr>
<tr>
<td>G</td>
<td>era (AD...)</td>
</tr>
<tr>
<td>C</td>
<td>century of era (&gt;=0)</td>
</tr>
<tr>
<td>Y</td>
<td>year of era (&gt;=0)</td>
</tr>
<tr>
<td>x</td>
<td>weekyear</td>
</tr>
<tr>
<td>w</td>
<td>week of weekyear</td>
</tr>
<tr>
<td>e</td>
<td>day of week</td>
</tr>
<tr>
<td>E</td>
<td>day of week (Tuesday, Tue...)</td>
</tr>
<tr>
<td>y</td>
<td>year</td>
</tr>
<tr>
<td>D</td>
<td>day of year</td>
</tr>
<tr>
<td>M</td>
<td>month of year (July, Jul, 07...)</td>
</tr>
<tr>
<td>d</td>
<td>day of month</td>
</tr>
</tbody>
</table>

**Examples**

@toISODate("22-05-1982","dd-MM-yyyy") → 1982-05-22

#### toISOTime

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Time&gt; @toISOTime(&lt;String&gt;,&quot;HH:mm:ss&quot;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Transforms a string to an ISO time, being first parameter the date you want to read and the second parameter the pattern to read it</td>
</tr>
<tr>
<td>a</td>
<td>halfday of day (AM~PM)</td>
</tr>
<tr>
<td>K</td>
<td>hour of halfday (0~11)</td>
</tr>
<tr>
<td>h</td>
<td>clockhour of halfday (1~12)</td>
</tr>
<tr>
<td>H</td>
<td>hour of day (0~23)</td>
</tr>
<tr>
<td>k</td>
<td>clockhour of day (1~24)</td>
</tr>
<tr>
<td>m</td>
<td>minute of hour</td>
</tr>
<tr>
<td>s</td>
<td>second of minute</td>
</tr>
<tr>
<td>S</td>
<td>fraction of second</td>
</tr>
</tbody>
</table>

**Examples**

@toISODate("10:25:37","HH:mm:ss") ✭ 10:25:37

@toISODate("10:25:37pm","hh:mm:ssaa") ✭ 22:25:37

---

### toISODateTime

**Use**

`<DateTime>@toISODateTime(<String>,"yyyy-MM-dd'T'HH:mm:ss")`

**Description**

Transforms a string to an ISO date time, being first parameter the date you want to read and the second parameter the pattern to read it

- **G**: era (AD...)
- **C**: century of era (>=0)
- **Y**: year of era (>=0)
- **x**: weekyear
- **w**: week of weekyear
- **e**: day of week
- **E**: day of week (Tuesday, Tue...)
- **y**: year
- **D**: day of year
- **M**: month of year (July, Jul, 07...)
- **d**: day of month
- **a**: halfday of day (AM~PM)
- **K**: hour of halfday (0~11)
- **h**: clockhour of halfday (1~12)
- **H**: hour of day (0~23)
| k | clockhour of day (1~24) |
| m | minute of hour |
| s | second of minute |
| S | fraction of second |

**Examples**

@toISODateTime("1982-05-22T10:25:37","yyyy-MM-dd'T'HH:mm:ss") ➤ 1982-05-22T10:25:37
@toISODateTime("19820522 102537","yyyyMMddHHmmss") ➤ 1982-05-22T10:25:37

---

**monthNameFromValue**

**Use**

<String>@monthNameFromValue(<Num>,<Num>)

**Description**

Returns the name of the month from a value (from 1 = January to 12 = December) and a length of the desired output string. Any 0 or negative length returns the full word.

**Examples**

@monthNameFromValue(1,2) ➤ “Ja”
@monthNameFromValue(2,3) ➤ “Feb”

---

**day-from-date**

**Use**

<Num> @day-from-date(<Date>)

**Description**

Extracts days value from an ISO Date.

**Examples**

@day-from-date(@toDate("1992-07-05")) ➤ 5

---

**day-from-dateTime**

**Use**

<Num> @day-from-dateTime (<DateTime>)

**Description**

Extracts days value from an ISO DateTime.

**Examples**

@day-from-dateTime(@toDateTime("1992-07-05T12:13:14")) ➤ 5
<table>
<thead>
<tr>
<th><strong>month-from-date</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>month-from-dateTime</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>year-from-date</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>year-from-dateTime</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>seconds-from-dateTime</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td>seconds-from-time</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minutes-from-dateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>minutes-from-time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hours-from-dateTime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><strong>Examples</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hours-from-time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
</tr>
</tbody>
</table>
### current-dateTime

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;DateTime&gt; @current-dateTime()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets current system datetime.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### current-date

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Date&gt; @current-date()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets current system date.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### current-time

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Time&gt; @current-time()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets current system time.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### current-dateTime-TZ

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;DateTime&gt; @current-dateTime-TZ()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets current system datetime with timezone.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### current-date-TZ

<table>
<thead>
<tr>
<th>Use</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>Use</td>
<td>&lt;Date&gt; @current-date-TZ ()</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------------</td>
</tr>
<tr>
<td>Description</td>
<td>Gets current system date with timezone.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### current-time-TZ

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Time&gt; @current-time-TZ ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Gets current system time with timezone.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### toDateTime

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;DateTime&gt; @toDateTime(&lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Transforms current String in ISO format to a DateTime.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### toDate

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Date&gt; @toDate(&lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Transforms current String in ISO format to a Date.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>

### toTime

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Time&gt; @toTime(&lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Transforms current String in ISO format to a Time.</td>
</tr>
<tr>
<td>Examples</td>
<td></td>
</tr>
</tbody>
</table>
### days-from-duration

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Num&gt; @days-from-duration (&lt;Duration&gt;)</td>
<td>Calculates the total number of whole days in a normalized duration value. This is not necessarily the same as the integer that appears before the D in the value. For example, if the duration is P1DT36H, the function returns 2 rather than 1. This is because 36 hours is equal to 1.5 days, and the normalized value is therefore P2DT12H. This function does not round the number of days; if the duration is 2 days and 23 hours, it returns the integer 2.</td>
<td>@days-from-duration(@current-dateTime() - /baby/birthdate) 153</td>
</tr>
</tbody>
</table>

### total-days-from-duration

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Num&gt; @total-days-from-duration (&lt;Duration&gt;)</td>
<td>Returns the total number of days in &lt;Duration&gt;. If it is an uneven number of days, it will return a fractional part.</td>
<td>@total-days-from-duration(@current-dateTime() - /baby/birthdate) 153.984</td>
</tr>
</tbody>
</table>

### hours-from-duration

<table>
<thead>
<tr>
<th>Use</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Num&gt; @hours-from-duration (&lt;Duration&gt;)</td>
<td>Calculates the hours component of a normalized duration value, as an integer between -23 and 23 (inclusive). This is not necessarily the same as the integer that appears before the H in the value. For example, if the duration is PT1H90M, the function returns 2 rather than 1. This is because 90 minutes is equal to 1.5 hours, and the normalized value is therefore PT2H30M. Likewise, if the duration is PT36H, the result is 12, because the normalized value is P1DT12H.</td>
<td>@hours-from-duration(/procedure/end - /procedure/start) 3</td>
</tr>
</tbody>
</table>

### total-hours-from-duration
<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @total-hours-from-duration (&lt;Num&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the total number of hours in Duration. If it is an uneven number of hours, it will return a fractional part.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@total-hours-from-duration(/procedure/end - /procedure/start)</code> =&gt; 3.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @minutes-from-duration (&lt;Duration&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Calculates the minutes component of a normalized duration value, as an integer between -59 and 59 inclusive. This is not necessarily the same as the integer that appears before the M in the value. For example, if the duration is PT1M90S, the function returns 2 rather than 1. This is because 90 seconds is equal to 1.5 minutes, and the normalized value is therefore PT2M30S. Likewise, if the duration is PT90M, the result is 30, because the normalized value is PT1H30M.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@minutes-from-duration(/fatiguetest/end - /fatiguetest/start)</code> =&gt; 23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @total-minutes-from-duration (&lt;Duration&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the total number of minutes in Duration. If it is an uneven number of minutes, it will return a fractional part.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@total-minutes-from-duration(/fatigueTest/end - /fatigueTest/start)</code> =&gt; 23.9843</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @seconds-from-duration (&lt;Duration&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Calculates the seconds component of a normalized duration value, as a decimal number between -60 and 60 exclusive. This is not necessarily the same as the number that appears before the S in the value. For example, if the duration is PT90S, the function returns 30 rather than 90. This is because 60 of those seconds are considered to be one minute, and the normalized value would therefore be PT1M30S.</td>
</tr>
</tbody>
</table>

99
<table>
<thead>
<tr>
<th>Examples</th>
<th>@seconds-from-duration(/coagulationTest/end - /coagulationTest/start)</th>
<th>12</th>
</tr>
</thead>
</table>

| **total-seconds-from-duration**              |                                                                      |    |
| Use                                          | <Num> @total-seconds-from-duration (<Duration>)                      |    |
| Description                                  | Returns the total number of seconds in Duration.                     |    |
| Examples                                     | @total-seconds-from-duration(/coagulationTest/end - /coagulationTest/start) | 12.5 |

### 9.4.7 Ontological functions

| **valueid**                                   |                                                                      |    |
| Use                                          | <String> @valueid()                                                   |    |
| Description                                  | This function introduces linkEHR internal id. Must be the only expression of the function. Can be useful to map the nodes like "rc_id". |    |
| Examples                                     |                                                                      |    |

| **uuid**                                      |                                                                      |    |
| Use                                          | <String> @uuid()                                                      |    |
| Description                                  | This function introduces generates an UUID. Can be useful to map the nodes like "rc_id". |    |
| Examples                                     | @uuid() "83d9c94a-cd88-4ca8-a6a8-dc0a240a7d98"                      |    |

| **node-id**                                   |                                                                      |    |
| Use                                          | <String> @node-id()                                                   |    |
| Description                                  | Returns a unique id created from the source structure path.          |    |
## code

**Use**  
`<String> @code()`  

**Description**  
Inserts the code from the ontology.

**Examples**  
(inside an attribute from Blood pressure archetype Systolic Element)  
@code() ➤ “at0001”

## archetypeName

**Use**  
`<String> @archetypeName()`  

**Description**  
Inserts the archetype identifier.

**Examples**  
(inside any attribute from Blood pressure archetype)  
@archetypeName() ➤ “CEN-EN13606-ENTRY.Blood_pressure.v1”

## text

**Use**  
`<String> @text()`  

**Description**  
Inserts the text of the nearest father object on the ontology for the default language.

**Examples**  
(inside an attribute from Blood pressure archetype Systolic Element)  
@text() ➤ “Systolic”

## text

**Use**  
`<String> @text("atXXXX")`  

**Description**  
Inserts the text of the object with the chosen node identifier on the ontology for the default language. If the code is left as “atXXXX” LinkEHR will replace it with the nearest node id to current attribute.
### text

**Use**  
`<String> @text(“atXXXX”,<Language>)`

**Description**  
Inserts the text of the object with the chosen code on the ontology for the chosen language. If the code is left as “atXXXX” LinkEHR will replace it with the nearest node id to current attribute.

**Examples**  
(inside any attribute from Blood pressure archetype)  
@text(“at0001”, "es") “Sistólica”

---

### nearest-text

**Use**  
`<String> @nearest-text()`

**Description**  
Inserts the nearest non-Reference Model text that can be found in the archetype.

**Examples**  
(inside an attribute from Blood pressure archetype Systolic Element)  
@nearest-text() “Systolic”

---

### description

**Use**  
`<String> @description()`

**Description**  
Inserts the description of the nearest father object on the ontology for the default language.

**Examples**  
(inside an attribute from Blood pressure archetype Systolic Element)  
@description() “The measurement by any means...”

---

### description

**Use**  
`<String> @description("atXXXX")`
<table>
<thead>
<tr>
<th>Description</th>
<th>Inserts the description of the object with the chosen node identifier on the ontology for the default language.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>(inside any attribute from Blood pressure archetype) @description(&quot;at0001&quot;) “The measurement by any means…”</td>
</tr>
</tbody>
</table>

**description**

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;String&gt; @description(&quot;atXXXX&quot;,&lt;Language&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Inserts the description of the object with the chosen code on the ontology for the chosen language.</td>
</tr>
<tr>
<td>Examples</td>
<td>(inside any attribute from Blood pressure archetype) @description(&quot;at0001&quot;, “es”) “La medida por cualquier método…”</td>
</tr>
</tbody>
</table>

**getDescription**

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;String&gt; @getDescription(&lt;String&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns a String with the description of a code from the terminology (SNOMED)</td>
</tr>
<tr>
<td>Examples</td>
<td>@getDescription(&quot;73211009&quot;) “Diabetes mellitus (disorder)”</td>
</tr>
</tbody>
</table>

**in**

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Bool&gt; @in(&lt;Code&gt;, &lt;Expression&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns a boolean stating if the code was found on the expression</td>
</tr>
<tr>
<td>Examples</td>
<td>@in(&quot;408539000&quot;, &quot;&lt;&lt; 73211009</td>
</tr>
</tbody>
</table>

**translate**

| Use | <String> @translate(<Code>, <Terminology>) |
# DESCRIPTION

<table>
<thead>
<tr>
<th>Description</th>
<th>Returns the translation of the provided code into the selected terminology or subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>@translate(&quot;ACT1&quot;,&quot;myLocalConceptMap&quot;) “123456”</td>
</tr>
</tbody>
</table>

## getgetDescriptionTerminology

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;String&gt; @getDescriptionTerminology (&lt;Code&gt;, &lt;Terminology&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the term corresponding to the provided code from the selected terminology or subset</td>
</tr>
<tr>
<td>Examples</td>
<td>@getDescriptionTerminology(&quot;73211009&quot;,&quot;SNOMED-CT&quot;) “diabetes mellitus”</td>
</tr>
</tbody>
</table>

## augment

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Object&gt; @augment (&lt;AugmentationID&gt;, &lt;SourcePaths&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Augments instance with the corresponding augmentation. E.g. Add a cluster with drug contraindication information extracted from paths</td>
</tr>
<tr>
<td>Examples</td>
<td>@augment(&quot;drugContraindicationId&quot;,&quot;/pathtodrug1, /pathtodrug2&quot;)</td>
</tr>
</tbody>
</table>

## Aggregation functions

### count

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @count(&lt;Path&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Counts the number of times the path is repeated on the source data.</td>
</tr>
<tr>
<td>Examples</td>
<td>@count(/problem/code) 2</td>
</tr>
<tr>
<td>Description</td>
<td>Counts the number of times the path is repeated (that apply to the given condition) on the source data. Condition and Path must be over the same branch.</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@count(/problem/code, @in(/problem/code,&quot;&lt;128462008&quot;))</code> 1</td>
</tr>
</tbody>
</table>

### sum

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @sum(&lt;Path&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sums the numeric values on the selected path.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@sum(/medication/dose)</code> 1344</td>
</tr>
</tbody>
</table>

### sum

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @sum(&lt;Path&gt;, &lt;Condition&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Sums the numeric values on the selected path that apply to the given condition.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@sum(/medication/dose, @in(/medication/ingredient,&quot;&lt;441900009&quot;))</code> 124</td>
</tr>
</tbody>
</table>

### max

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @max(&lt;Path&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the maximum value on the selected path.</td>
</tr>
<tr>
<td>Examples</td>
<td><code>@max(/bloodpressure/systolic)</code> 116</td>
</tr>
</tbody>
</table>

### max

<table>
<thead>
<tr>
<th>Use</th>
<th><code>&lt;Num&gt; @max(&lt;Path&gt;, &lt;Condition&gt;)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the maximum value on the selected path that applies to the given condition.</td>
</tr>
</tbody>
</table>
## min

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @min(&lt;Path&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the minimum value on the selected path.</td>
</tr>
<tr>
<td>Examples</td>
<td>@min(/bloodpressure/systolic) 95</td>
</tr>
</tbody>
</table>

## min

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @min(&lt;Path&gt;, &lt;Condition&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the minimum value on the selected path that applies to the given condition.</td>
</tr>
<tr>
<td>Examples</td>
<td>@max(/bloodpressure/systolic,/bloodpressure/date&gt;@toISODate(&quot;20170101&quot;)) 97</td>
</tr>
</tbody>
</table>

## avg

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @avg(&lt;Path&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the average for the numeric values on the selected path.</td>
</tr>
<tr>
<td>Examples</td>
<td>@avg(/vital/weight) 87.85</td>
</tr>
</tbody>
</table>

## avg

<table>
<thead>
<tr>
<th>Use</th>
<th>&lt;Num&gt; @avg(&lt;Path&gt;, &lt;Condition&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Returns the average for the numeric values on the selected path that apply to the given condition.</td>
</tr>
<tr>
<td>Examples</td>
<td>@avg(/vital/weight,/vital/date&gt;@toISODate(&quot;20170101&quot;)) 86.4</td>
</tr>
</tbody>
</table>
9.4.9 Constraint values

If the path on the archetype where the mapping is being defined has their values constrained (e.g. only a specific set of strings is allowed in the archetype) LinkEHR shows them as a list in order to easily put them as mapping functions.

9.4.10 Data source access

Data source tab allows the definition of paths to the currently selected data source. Depending if the source is an XML Schema or an archetype different functionality apply. In both views a filter can be applied to show only the fields from the data source that follow it. In addition to that, if source schema is an archetype then a check to show the full archetype tree can be selected in order to map attributes and elements from the source archetype from the underlying reference model not included in it.
If no data source has been selected yet, the message “No data source selected” is displayed on the tab and no paths can be selected.

### 9.5 Advanced mapping

#### 9.5.1 Object mapping

In addition to attribute mappings, object builder can be attached to archetype node objects with multiple occurrences. An object builder comprises a source path and a condition. Object builders control the creation of target instances, in such a way that a new target instance is constructed for each source element addressed by the path that satisfies the filter. When no object builder
is defined default semantics is applied, which groups together instances that share the same context following a minimum-cardinality approach, i.e. it is just generated the minimum number of elements necessary for the result to be compliant with the archetype. Variables can be created in order to reuse common data sources paths, as seen in Figure 44.

![Add variable](image)

Figure 44. Adding a variable

The condition expression is edited with the same editor as the attribute mappings (see “Attribute mapping definition”) with the only difference of the data source paths, which are now dependant of the defined variables.

On this window is also possible to say if an object mapping has bag semantics and which is the parent object mapping of this object mapping (mapping without target paths can be parents of an object mapping, which allow reuse of object mappings).
9.5.2 Virtual node mapping

Any object in the archetype can be selected to generate a virtual node for mapping. These virtual nodes allow having more than one path from the source mapped to a single object in the archetype (e.g. mapping a source “past medication” and “current medication” into a “medication list” in the archetype. Once the virtual node is created, it can be mapped in the same way as the original one. Virtual nodes are represented by a green cube in the archetype tree. These nodes only exist in mapping view, going back to archetype edition will hide this nodes (be aware that modifying the archetype once virtual nodes have been created can make some virtual nodes not accessible and will be removed when saving)

9.5.3 Mapping reuse

Sometimes mapping definition with standards can be tedious, as same kinds of structures tend to be reused in several parts of the archetype, and context information is usually maintained. To ease this process, two main mechanisms are provided: Favorite mappings (which can copy
mapping tables and keep them into an specific view that will be kept even if the application is closed) and propagate mappings. “Propagate mappings” allows you to copy the mapping definition of a given subtree into an equivalent subtree of the same type. This process tries to mimic both structures by expanding internal references if possible.

9.5.3.1 Internal references
Internal references can be deployed in mapping definition time. If only one internal reference is available in a given position, the internal reference is automatically expanded.

9.5.3.2 Archetype slots
Archetype slots can be left as is in mapping definition. This assumes that included archetypes have also been mapped to the same data source. If that is the case, when going to “Archetype transformation export” will allow you to choose which specific archetype you want to use in a given archetype slot. Be aware that archetypes must be compatible with the includes/excludes of the archetype slot in order to be able to select them.
10 Generation of archetype transformation programs

Once we have defined mapping at least for all mandatory paths (the root of the archetype must be coloured in green) we can generate automatically the transformation script by clicking the button of Export Integration Archetype.

NOTE: To save your mappings you must save the archetype explicitly by using the menu Archetype ▸ Save ▸ Save Archetype or the Save Archetype button. Mappings are saved in a file named as the archetype but with the .MAP extension that must reside in the same folder of the archetype .ADL file.

The export integration archetype window includes several parts and options.

- Valid covers: A cover archetype is a valid combination of attribute mappings that can generate instances of data. In other words, when we define mappings for different data sources in the same archetype, different covers will be generated (one for each possible combination of sources). We can review these covers and select the ones we
consider valid for our purposes. If several covers are left selected, a different XQuery will be generated for each one, but the application of these different queries will always return valid data instances with regard to the archetype and reference model definition. If only one data source is used in the mappings, only one cover will appear and must be left selected.

- Validate mapping: Shows a result of a validation check of the mapping.

- Preview XQuery (Figure 47): Shows a window with the generated XQuery code that can be copied and used elsewhere.

- Test XQuery (Figure 48): We can select a test XML data file (which must be an instance of the data source schema used for defining the queries) and execute the generated query in order to see the resultant normalized XML document (Figure 49).

- Language: Choose the language to generate the transformation script. This effectively makes the language of the archetype the default one for the use of ontological functions.

- Export Integration Archetype: This option generates an .LKR file which includes the generated scripts for the integration archetype.

- Export Template: This option exports the template (or archetype) to a given file.

**NOTE:** The result of using LinkEHR is the generation of .LKR files. These files contain all the necessary information to access, extract and transform legacy data into archetyped data instances; and will be the input for LinkEHR Integration Engine. This module is a data server which serves normalized instances of data or standard EHR extracts from the original data sources.

The generated transformation script can be used by any Information System with XQuery execution capabilities. LinkEHR uses the open source engine Saxon-B from http://www.saxonica.com/
Please note that some transformation functions used at the generated XQuery will require the use of third-party libraries, such as the Joda-Time library for ISO dates manipulation. [http://joda-time.sourceforge.net/]
Selection of covers

If a given archetype node can be generated in more than one way (e.g. has been mapped to two different sources) more than one cover will be presented in the Export integration archetype dialog, as more than one transformation program can be created (each one with one of the alternative mappings). This can become a problem as possible transformation programs explode in a combinatorial way (e.g. 4 nodes with 2 alternatives each would generate $2^4$ combinations).
In principle, each one of these covers is a correct way of generating a normalized instance, so covers are not presented in any particular order. LinkEHR allows you to review each one of these covers so you can see which combination of mappings are included, and then choose if you want to export it or not by selecting the corresponding check in the “Select valid covers” column.

10.2 Skolem function edition

A skolem function returns a different value (identifier) for each combination of parameters. At any given level, a new instance of that object will be only created if another instance with the same skolem identifier does not exist. LinkEHR calculates automatically the skolem functions based on the identifiers calculated in the tree hierarchy. This skolem functions can be changed by removing some of the identifiers detected in the automatic process. The following capture of the Export integration archetype dialog shows the Skolem function editor part.

10.3 Testing the archetype transformation program

With “Test XQuery” a user can test the results from a given transformation program. This is done by selecting an XML file and optionally a patient id (for sources that contain more than one patient, a single transformation program can generate several records). In case of an XSD was provided when defining a data source, XML file will be tested against the schema, and show a warning if the XML is not compliant with it. Testing result allows you to view an html preview of data (available for the models that have an XSLT visualization file defined in the documentation), an XML, and a json outputs.

10.4 Exporting the archetype transformation program

Pressing the “Export Integration archetype” button generates a LKR file containing the archetype and all selected cover’s XQuery scripts. It also includes the mapping information, the defined custom rules, and if available, the XML Schema of the data source. This LKR file can be used as is on a different LinkEHR Studio installation, as it contains all the information needed to continue with the mapping process.

10.5 Using the archetype transformation program outside LinkEHR Studio

To execute the XQuery transformation program an XQuery library is needed. For java, at least saxon 8.9 or newer should be used. This is the minimum version that can execute java classes from XQuery, which is used by LinkEHR for some complex function support.
11 Troubleshooting

General

- LinkEHR does not run
  - LinkEHR is provided in both 32 and 64 bit versions. Please ensure that the LinkEHR you are trying to use is compatible with your current installed Java VM. You can see which one is your current Java VM (JVM) by opening a command window and input “java –version”.
  - Not having a Java VM installed. Please open a command window and input “java –version”. If there is an error then you need to download and install a JVM, or at least put it in the path.

Edition

- When I open an archetype, nothing happens
  - Check Console view for information on the parsing errors.

Mapping

- I don’t get any results
  - Check occurrences and cardinality of the nodes, as some combinations do not really allow instances to have contents.
  - Alternatively you can try the “Test current branch” method, that will help you in knowing which branches are actually incorrect
  - Check if the selected data instance is really compatible with the supposed schema.
Annex 1. The ADL language

See https://specifications.openehr.org/releases/AM/latest/ADL1.4.html for a latest version of ADL1.4 specs
Annex 2. CEN/ISO 13606 specific editor

To activate CEN EN13606 editor you must activate the option “Use specific editor when available” under the Visualization tab on LinkEHR configuration dialog (see Figure 4). After doing this, whenever you create or edit a CEN EN13606 archetype the specific editor will also be opened.

Choosing a root entity for a CEN EN13606 archetype

Before starting the proper archetype edition, it is necessary to choose which will be the root entity of your archetype. CEN EN13606 provides six different entities:

- FOLDER: A level of organization used to group information on clinical episodes, problems, time periods or clinical service
- COMPOSITION: A complete clinical document in the scope of a clinical session.
- SECTION: EHR data that are inside of a composition and comes from a single Figure process
- ENTRY: A clinical fact that is registered.
- CLUSTER: Groups data into lists or tables.
- ELEMENT: Container of the leaf nodes of an EHR structure.

Usually, root class is a COMPOSITION or ENTRY. Figure 50 shows the archetype class selection algorithm for CEN EN13606
Archetype definition class selection algorithm for CEN EN13606
Adapted from openEHR
Editing CEN EN13606 archetypes with LinkEHR

By creating a new archetype (selecting ‘CEN’ as organization and ‘EN13606’ as reference model) the archetype tree view will look like this (here we have chosen a COMPOSITION with name “document test”, note that root icon will change if a different root entity is chosen).

A newly created composition

From there, clicking on a node shows its properties. Note that “ANY ALLOWED” is checked, so you must uncheck it to change its properties right away (but if you keep editing the archetype tree, LinkEHR will take care of this for you).
On CEN EN13606 editor, each one of the reference model classes has its own view to change its properties. Every view has three things in common: The “ANY ALLOWED” (if this checkbox is checked means that no further restrictions will be applied to this node of the archetype), the “Occurrences”, which mean how many distinct objects of this kind will appear on the data, (e.g. A patient can have one or two different telephone numbers so one ‘telephone number’ node can be created with minimum occurrences of one and maximum occurrences of two will be enough to express that restriction), and lastly the “Ontology” section, where a text, a description and binding can be set in order to make richer archetypes.

Each one of CEN EN13606 classes has also a part of the form to change how many children can have. On FOLDER class it is called sub-folders, on COMPOSITION class it is called content, on SECTION class it is called members, on ENTRY class it is called items and on CLUSTER class it is called parts.
Other additional properties can be changed for each one of the classes available on the reference model.

Next step is creating a child for the root object. This is done for the node menu by either right clicking on a node or pressing the “Node...” button on the toolbar. This will pop up a menu like the one in Figure 53.

Selecting one of the alternatives will add this node to the archetype tree.

- **CEN EN13606 Datatypes**

Datatypes are the leaf nodes of the archetype tree on CEN EN13606 editor. Each one of the data types is intended to be used for a specific purpose.

- **BL**: Boolean data value. Value data type that can be true or false.
- **CODED_TEXT**: A free text string with an associated coded value.
- **CV**: Coded Value. Coded data, stating only a code, without classifiers or translations to other codification systems.
- **DATE**: Identifies a single day of calendar, expressed by a combination of calendar year, calendar month, calendar week, calendar day or day of the year.
● DURATION: A period of time from a non-fixed set of time (which is not specified). It can be expressed as a negative duration, meaning that duration is backwards. It must not be used to express points of time (for this, use TS)

● ED: Encapsulated data. Data which main purpose is beign interpreted by humans. This data includes any written language, multimedia data, digital signatures or information defined in any other standard.

● IVL: Interval. An interval in CEN EN13606 can contain PQ, TS, DURATION, ORD and RTO. It is defined with a start and end object of the selected data type. As is widely used, the interval of time has its own type (IVLTS).

● IVLTS: Interval time. Can be defined by a start date or time and an end date or time, by a duration without start or end, by a start date or time and a duration, and by a duration and an end date or time. Do not confuse with DURATION

● ORD: Ordinal. A number that defines a position in a list or series with a textual description.

● PQ: Physical quantity. A dimensioned quantity that expresses the result of a measurement.

● RTO: Ratio. A quantity built as the quotient of a numerator quantity divided by a denominator quantity.

● SIMPLE_TEXT: A simple text without an associated code.

● TS: Time point. A non-dimensional time moment.

● URI: Universal resource identifier. A telecom address as specified in standard RFC 1738
Annex 3. OpenEHR specific editor

As in CEN EN13606 editor, to activate openEHR editor you must activate the option “Use specific editor when available” under the Visualization tab on LinkEHR configuration dialog (see Figure 4). After doing this, whenever you create or edit an openEHR archetype the specific editor will also be opened. It is worth noticing that in principle this guide is developed for an openEHR 1.0.2 reference model, but thanks to the reference model independence of LinkEHR Studio, newer openEHR reference model versions should be directly supported.

11.1 Choosing a root entity for an openEHR archetype

OpenEHR provides a wider range of classes that we can use as a root class in an archetype. Both EN13606 and openEHR provide similar entities, the main difference being that openEHR provides several ‘entry’-level and ‘cluster’-level entities. OpenEHR ‘entry’-level entities are usually less generic and have specific meaning per-se.

With all this, openEHR contains the following main root classes:

- **FOLDER**: A level of organization used to group information on clinical episodes, problems, time periods or clinical service
- **COMPOSITION**: A complete clinical document in the scope of a clinical session.
- **SECTION**: EHR data that are inside of a composition and comes from a single process
- **Clinical statement classes**:
  - **OBSERVATION**: Clinical observations or evidence, such as measurements, examination findings or information provided by the patient such as symptoms.
  - **EVALUATION**: Statements used to capture clinically interpreted findings, opinions and summary statements. Usually the clinician’s interpretation of observations.
  - **INSTRUCTION**: Statements about what should happen in the future, e.g. medication or clinical order
  - **ACTION**: Statements about what was done, e.g. medication administration
  - **ADMIN_ENTRY**: Statements for recording administrative events, e.g. admission, discharge, consent, etc.
- **CLUSTER**: Groups data into lists or tables.
- **ITEM_SINGLE**: Logical single value data structure. Used to represent any data which is logically a single value, such as a person’s height or weight.

- **ITEM_LIST**: 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.

- **ITEM_TABLE**: Logical relational database style table data structure, in which columns are named and ordered with respect to each other.

- **ITEM_TREE**: Logical tree data structure. Used for representing data which is logically a tree such as audiology, microbiology, or biochemistry results.

- **ELEMENT**: Container of the leaf nodes of an EHR structure.

As several entities share meaning with their EN13606 counterparts, Figure 50 is still a good algorithm for archetype class selection algorithm for openEHR. However, the ‘entry’-level classes contain additional semantics. An algorithm to choose the correct openEHR entry entity can be seen in Figure XX
Editing openEHR archetypes with LinkEHR

By creating a new archetype (selecting ‘openEHR’ as organization and ‘EHR’ as reference model) the archetype tree view will look like this (here we have chosen a COMPOSITION with name “document test”, note that root icon will change if a different root entity is chosen).

![Archetype Tree](image)

A newly created openEHR composition

From there, clicking on a node shows its properties. Note that “ANY ALLOWED” is checked, so you must uncheck it to change its properties right away (but if you keep editing the archetype tree, LinkEHR will take care of this for you).

![openEHR COMPOSITION properties form](image)

On openEHR editor, each one of the reference model classes has its own view to change its properties. Every view has three things in common: The “ANY ALLOWED” (if this checkbox is checked means that no further restrictions will be applied to this node of the archetype), the
“Occurrences”, which mean how many distinct objects of this kind will appear on the data, (e.g. A patient can have one or two different telephone numbers so one ‘telephone number’ node can be created with minimum occurrences of one and maximum occurrences of two will be enough to express that restriction), and lastly the “Ontology” section, where a text, a description and binding can be set in order to make richer archetypes.

Each one of openEHR classes has also a part of the form to change how many children can have. On FOLDER class it is called folders, on COMPOSITION class it is called content and on SECTION class it is called items. A difference between openEHR specific editor and EN13606 specific editor is that classes such as Observation make use of two different attributes for the definition of structure constraints (namely, data and protocol)

Other additional properties can be changed for each one of the classes available on the reference model.

Next step is creating a child for the root object. This is done for the node menu by either right clicking on a node or pressing the “Node…” button on the toolbar. This will pop up a menu like the one in the following figure.

Creating a new object on the tree

Selecting one of the alternatives will add this node to the archetype tree.

openEHR Datatypes
Datatypes are the leaf nodes of the archetype tree on openEHR editor. Each one of the data types is intended to be used for a specific purpose. OpenEHR specific editor contains most usual datatypes used in currently available openEHR archetypes. If other datatypes are needed they can be created in the Archetype Tree view in LinkEHR Studio.

- **DV_BOOLEAN**: Boolean data value. Value data type that can be true or false.
- **DV_CODED_TEXT**: A free text string with an associated coded value.
- **DV_COUNT**: Datatype used to constraint countable quantities (no units).
- **DV_DATE**: Identifies a single day of calendar, expressed by a combination of calendar year, calendar month, calendar week, calendar day or day of the year.
- **DV_DATE_TIME**: Time point. A non-dimensional time moment.
- **DV_TIME**: Identifies a time in an unspecified calendar day, expressed by a combination of hour, minutes, and seconds.
- **DV_DURATION**: A period of time from a non-fixed set of time (which is not specified). It can be expressed as a negative duration, meaning that duration is backwards. It must not be used to express points of time (for this, use DV_DATE_TIME).
- **DV_MULTIMEDIA**: Encapsulated data. Data which main purpose is being interpreted by humans. This data includes any written language, multimedia data, digital signatures or information defined in any other standard.
- **DV_IDENTIFIER**: Datatype for representing identifiers of real world entities.
- **DV_PARSABLE**: Encapsulated data expressed as a parsable string.
- **DV_ORDINAL**: Ordinal. A number that defines a position in a list or series with a textual description.
- **DV_QUANTITY**: Physical quantity. A dimensioned quantity that expresses the result of a measurement.
- **DV_PROPORTION**: Ratio. A quantity built as the quotient of a numerator quantity divided by a denominator quantity.
- **DV_TEXT**: A simple text. It can be derived to a coded_text.

**Integrating LinkEHR in an openEHR workflow**

LinkEHR is a multi-reference model archetype editor which allows working with archetypes based on any Reference Model (including openEHR, EN ISO 13606, HL7 CDA or HL7 FHIR). The
openEHR Archetype Editor, Ocean Template Editor and the Clinical Knowledge Manager (CKM) are implemented to support the openEHR RM only. This leads to small differences in the definition of archetypes.

In most cases, these differences do not affect to the complete compatibility between tools. In some specific cases, an export process is available in LinkEHR to create archetypes compatible with openEHR legacy tooling.

Finally, it is worth mentioning that several of the described incompatibilities have been addressed in the future AOM/ADL 2.0 specifications, currently in development, which are closer to the original LinkEHR Editor design decisions.

**Summary of different implementation approaches**

**Node identifiers**

According to the AOM/ADL 1.4 specifications, all Object nodes of an archetype can have a node identifier (atNNNN code) that allows associating a name, description and a terminology code to the node. LinkEHR Editor creates these identifiers by default in all nodes, while openEHR Editor only creates it for sibling nodes that are also archetypable (locatable).

It must be noted that the future AOM/ADL 2.0 specifications has adopted the LinkEHR Editor approach, by assigning a node identifier to all object nodes.

In some cases, openEHR Editor or Template Designer may fail while loading these additional identifiers, due an incomplete implementation of the specifications.

**Solutions**

- An export process can be executed in LinkEHR to create compatible archetypes with the openEHR tools.
Support of domain types

Domain types are custom archetype constraints defined for a particular Reference Model. They simplify the definition of some common constraints, although they are completely equivalent to defining archetype constraints following the generic model.

Thus, openEHR has defined three custom domain types, namely C_CODE_PHRASE, C_DV_ORDINAL and C_DV_QUANTITY. Available openEHR archetypes make extended use of these classes.

LinkEHR specific openEHR editor supports the edition of openEHR profile classes.
Editing an openEHR C_DV_QUANTITY

You can transform these openEHR profiles at any time by right clicking the domain type node and selecting either ‘Expand Domain Type’ or ‘Expand All Domain Types’.

Constraint bindings

A constraint binding (represented in ADL as an [acNNNN] code) allows the definition of value sets in archetypes based on an external terminology. According to the AOM/ADL 1.4 specifications, these bindings can be attached to any object node of the archetype. However, the specifications are incomplete about the exact use of this type of constraints.

Currently, LinkEHR Editor supports the definition of constraint bindings in two different points of the archetype. The first one, as a substitute of primitive String object nodes. That means that, at any point of the archetype where a String node can be defined, a constraint binding can be
defined instead. The second one as a substitute of a coded data type, if it has been previously configured in the RM documentation. For example, instead of an openEHR CODE_PHRASE data type or instead of an EN ISO 13606 CV or CD data types. Moreover, LinkEHR Editor supports the definition of explicit enumerations of terminology codes and multiple bindings per terminology.

openEHR Archetype Editor only supports the definition of constraint bindings as a substitute of the CODE_PHRASE data type. Moreover, only one binding per terminology can be defined in each node.

Thus, some incompatibility problems may occur when sharing archetypes between LinkEHR and openEHR tooling.

Solutions

- An export process can be implemented where only the first binding of each terminology is maintained for each different constraint binding. However, this will result in a loss of information of the model.

- An export process can be implemented to remove constraint bindings associated to primitive Strings. However, this will result in a loss of information of the model.

Constraining of abstract classes

The openEHR archetype Editor allows the definition of constraints over abstract classes (i.e. a constraint over an EVENT class instead of using POINT_EVENT or INTERVAL_EVENT children classes). This decision comes from the certainty that an archetype, when used in an end-user environment, has to include a precise definition of the structure of data. Stored data instances cannot make use of abstract classes. Instead leaving the decision of using an abstract class to the final user, we promote that that decision is taken by the clinical modeler.

Solutions

- When opening an openEHR archetype that includes an abstract class, the validator included in LinkEHR Editor can detect a problem in the archetype. The user can then right-click the abstract class and choose to convert it into one of its possible children classes.
openEHR-EHR-OBSERVATION.lab_test.v1

Definition

OBSERVATION [a0000] - Laboratory test [1]
  data (Obligatory)
  HISTORY [a0001] - Event Series [0..1]
    events [0..*] (Optional)
      EVENT [a0002] - Any event [0..*]
        data (Obligatory)
        ITEM_TREE [a0003] - Tree
  protocol (Optional)
  ITEM_TREE [a0004] - Tree [0..1]
  items [0..*] (Optional)
Choosing a root entity for a HL7 CDA archetype

Before starting the proper archetype edition, it is necessary to choose which will be the root entity of your archetype. HL7 CDA provides six different entities:

- **CLINICALDOCUMENT**: this element typically contains all of the namespace declarations necessary for the clinical document; may declare the realm for which the document was written; always indicates the version of CDA in use; and may declare the business rules that this document asserts conformance to.
- **AUTHOR**: represents the first of three different kinds of information sources from the HL7 RIM that appear in the CDA, and is arguably the most important since there must be at least one author in a CDA instance.
- **CUSTODIAN**: the custodian association links the assigned custodian to the clinical document. The assigned custodian is the organization that has been assigned the role to be the steward of the clinical document.
- **PARTICIPANT**: this kind of participant allows CDA to record associate other roles with a clinical document to support use cases not originally anticipated.
- **SECTION**: EHR data that are inside of a composition and comes from a single Figure process.
- **ENTRY**: A clinical fact that is registered.
- **ACT**: represents any kind of clinical act.
- **PROCEDURE**: this class is also very much like an act. It’s an act whose outcome results in the physical alteration of the subject.
- **ENCOUNTER**: is used to describe acts that are clinical encounter between a healthcare provider and a patient.
- **OBSERVATION**: is very similar to the procedure class and can be thought of as a “non-altering” procedure that results in a value.
- **OBSERVATIONMEDIA**: is a restricted form of the observation class intended to support other forms of media, such as an imaging result.
- **ORGANIZER**: is a specialization of the act class that is designed to support grouping of information.
- **SUBSTANCEADMINISTRATION**: is very similar to the procedure class, but has additional class attributes and associations to address medication specific information.
- **SUPPLY**: this class is used to describe things that are given to the patient for their subsequent use, possibly, as in the case of medications, for later administration.

Usually, root class is a CLINICALDOCUMENT. Figure 1 shows the dialog form to select the root element.
Selecting a CDA document.

Archetype Editor
This wizard creates the minimal archetype for the chosen reference model, standard and concept

| Organization: | HL7 |
| Reference Model: | CDA |
| Entity: | ClinicalDocument |
| Concept: | document_test |
| Language: | en-gb English (United Kingdom) |

Selecting a CDA document.

○ Editing HL7 CDA archetypes with LinkEHR-Ed
By creating a new archetype (selecting ‘HL7’ as organization and ‘CDA’ as reference model) the archetype tree view will look like this (here we have chosen a ClinicalDocument with name “document_test”, note that root icon will change if a different root entity is chosen).

A newly created ClinicalDocument

From there, clicking on a node shows its properties. Note that “ANY ALLOWED” is checked, so you must uncheck it to change its properties right away (but if you keep editing the archetype tree, LinkEHR-Ed will take care of this for you).
On HL7 CDA editor, each one of the reference model classes has its own view to change its properties. Every view has three things in common: The “ANY ALLOWED” (if this checkbox is checked means that no further restrictions will be applied to this node of the archetype), the “Occurrences”, which mean how many distinct objects of this kind will appear on the data, (e.g. A clinic act can have one or two different observations, so one observation node can be created with minimum occurrences of one and maximum occurrences of two will be enough to express that restriction), and lastly the “Ontology” section, where a text, a description and binding can be set in order to make richer archetypes.

Other additional properties can be changed for each one of the classes available on the reference model.
Next step is creating a child for the root object. This is done for the node menu by either right clicking on a node or pressing the “Node…” button on the toolbar. This will pop up a menu like the one in Figure 53.

Creating a new object on the tree

Selecting one of the alternatives will add this node to the archetype tree.

- **HL7 CDA Datatypes**

Datatypes are the leaf nodes of the archetype tree on HL7 CDA editor. Each one of the data types is intended to be used for a specific purpose.

- **BL**: Boolean data value. Value data type that can be true or false.
- **INT**: Integer data represents a positive or negative integer or zero. Negative integers are preceded with a minus sign. The integer (INT) data type is used sparingly in the CDA schema.
- **REAL**: the real data type follows from INT, save that it contains positive or negative real numbers or zero, rather than just integers. Appear in the value attribute as either decimal numbers or double precision floating point numbers in XML.
- **ST**: The String data is perhaps the easiest to understand. It encodes simple text data.
- **CD**: the Concept Descriptor, in addition to several attributes describing the code, it can contain a reference to the original text.
- **CE**: the Coded with Equivalents type is used to exchange coded concepts that are not permitted to contain qualifiers and so do not allow for codes to be created compositionally using post-coordination.
● CS: the Coded Simple type is used to convey codes that have a fixed value for codeSystem.
● CV: the Coded Value data type derives from the CE data type. It is used when only an unqualified coded concept without translations is desired in an exchange.
● II: the Instance Identifier data type is used to identify different instances of a kind of thing. It is used extensively in the CDA specification to identify persons, places, things, actions, roles, etc.
● ED: Encapsulated data is the way that HL7 transmits data in formats not defined by HL7. This data can include images, video, audio, etc. The data may be referenced, or the data may be directly incorporated into the CDA document.
● IVL_Ts: Interval of time data is often used to record a time interval over which some observation or event occurred or is intended to occur. Intervals of time are be specified using at most any two of the following components.
● TS: a time stamp is an instant time. Since this is implicitly a quantity of time since some arbitrarily chosen epoch of time, it is part of the quantity hierarchy. The HL7 standard does not define an epoch date to be used since a system can use any epoch value and still process time stamps correctly.
● PQ: Physical quantity. A dimensioned quantity that expresses the result of a measurement.
● RTO: Ratio. A quantity built as the quotient of a numerator quantity divided by a denominator quantity.
● URL: Universal Resource Locator. A telecom address as specified in standard RFC 1738
## Annex 5. Licenses

The following open source and free software libraries are used by LinkEHR Studio.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>License</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax-activation</td>
<td>1.1</td>
<td>CDDL 1.0</td>
<td>Used for the mail sending</td>
</tr>
<tr>
<td>dk-briks-automaton</td>
<td>1.0</td>
<td>BSD 2-clause</td>
<td>Dependency for xeger library</td>
</tr>
<tr>
<td>commons-beanutils</td>
<td>1.8.3</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-codec</td>
<td>1.7</td>
<td>Apache 2.0</td>
<td>Url and base64 encoding</td>
</tr>
<tr>
<td>commons-collections</td>
<td>3.2.1</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-configurations</td>
<td>1.5</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-dbcp</td>
<td>1.2.2</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-io</td>
<td>1.4</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-lang</td>
<td>2.4</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>commons-logging</td>
<td>1.1</td>
<td>Apache 2.0</td>
<td>Error log</td>
</tr>
<tr>
<td>commons-math3</td>
<td>3.6.1</td>
<td>Apache 2.0</td>
<td>Data analysis for mapping</td>
</tr>
<tr>
<td>commons-pool</td>
<td>1.3</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>dom4j</td>
<td>1.6.1</td>
<td>BSD</td>
<td>XML processing</td>
</tr>
<tr>
<td>ezmorph</td>
<td>1.0.6</td>
<td>Apache 2.0</td>
<td>JSON library dependency</td>
</tr>
<tr>
<td>apache-geronimo</td>
<td>1.1.1</td>
<td>Apache 2.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>hibernate</td>
<td>3.3</td>
<td>LGPL 2.1</td>
<td>Data access (deprecated code)</td>
</tr>
<tr>
<td>hsqldb</td>
<td>1.8.0</td>
<td>BSD</td>
<td>Hibernate dependency</td>
</tr>
<tr>
<td>jaxen</td>
<td>1.1</td>
<td>Apache 2.0</td>
<td>XPath execution</td>
</tr>
<tr>
<td>jdom</td>
<td>1.1</td>
<td>Apache 2.0</td>
<td>XML manipulation</td>
</tr>
<tr>
<td>jersey</td>
<td>1.9</td>
<td>GPL 2.0/CDDL1.1</td>
<td>Web services</td>
</tr>
<tr>
<td>joda-time</td>
<td>1.2.1</td>
<td>Apache 2.0</td>
<td>ISO time transformation</td>
</tr>
<tr>
<td>json-lib</td>
<td>2.4</td>
<td>Apache 2.0</td>
<td>JSON support</td>
</tr>
<tr>
<td>Library</td>
<td>Version</td>
<td>License</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>---------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>junit</td>
<td>3.8.1</td>
<td>EPL 1.0</td>
<td>Dependency for several libraries</td>
</tr>
<tr>
<td>log4j</td>
<td>1.2.9</td>
<td>Apache 2.0</td>
<td>Error log</td>
</tr>
<tr>
<td>javax-mail-api</td>
<td>1.3.2</td>
<td>GPL 2.0/CDDL1.1</td>
<td>Mail sending</td>
</tr>
<tr>
<td>mimepull</td>
<td>1.6</td>
<td>GPL 2.0/CDDL1.1</td>
<td>Jersey dependency</td>
</tr>
<tr>
<td>poi</td>
<td>3.9</td>
<td>Apache 2.0</td>
<td>Access to Microsoft format files</td>
</tr>
<tr>
<td>saxon</td>
<td>8.9</td>
<td>MPL 1.0</td>
<td>XSLT and XQuery processor</td>
</tr>
<tr>
<td>uuid</td>
<td>3.2</td>
<td>MIT</td>
<td>UUID generation</td>
</tr>
<tr>
<td>nl-flotsam-xeger</td>
<td>1.0</td>
<td>Apache 2.0</td>
<td>Sample string generation from regex</td>
</tr>
</tbody>
</table>