Multi-Level Health Information Modelling

MLHIM

**Reference Man-
ual**

Release 2.4.4

PRE-RELEASE

(planned release date is 2014-06-01)

The goal of MLHIM is to be Minimalistic, Sustainable, Implementable AND Interoperable.

Table of Contents

Front Matter

Acknowledgements

This work has received financial and in-kind support from the following persons and organizations:

- National Institute of Science and Technology on Medicine Assisted by Scientific Computing (INCT-MACC), coordinated by the National Laboratory of Scientific Computing (macc.lncc.br)
- Multilevel Healthcare Information Modeling Technological Development Unit, Member of the Emergent Group for Research and Innovation in Healthcare Information Technology, coordinated by Prof. Luciana Tricai Cavalini, MD, PhD (lutricav@mlhim.org)
- Timothy W. Cook, Independent Consultant
- Roger Erens, Independent Consultant

Using the MLHIM Reference Manual

This section describes typographical conventions and other information to help you get the most from this document.

The intended audience for this manual includes; software developers, systems analysts and knowledge modelers in the healthcare domain. It is assumed that the reader has knowledge of object-oriented notation, concepts and software construction practices.

In the PDF release of these specifications cross reference links are denoted by a number inside square brackets i.e. [5].

Release Information

The official published version is in PDF format in the English language. The ODT version is always considered a work in progress. Each version is labeled with the release number of the reference model and its implementation in XML Schema 1.1.

Previous versions of the PDF release had a version number that is the date of release followed by the language code and locality code. As an example, a release in English on January 1, 2011 will have the filename: mlhim-ref-man-2011-01-01-en-US.pdf

Error Reporting

Please report all errors in documentation and/or in the specifications of the information model as bug reports at the GitHub development site. It is easy to do and a great way to give back. See: MLHIM Issues

Pronunciation

MLHIM is pronounced 'muh-leem Click Hear How It Sounds for when it is used in spoken English language such as presentations or general discussions.

Purpose

The purpose of the MLHIM project is to provide a free and openly available specification and implementation of an interoperability solution for healthcare information exchange. The MLHIM specifications are designed to be fully independent of any implementation specific contexts. Therefore workflow, security, user access, data persistence, etc. are all outside the scope of MLHIM.

Conformance

Conformance to these specifications are represented in a Language Implementation Specification (LIS). A LIS is formal document detailing the mappings and conventions used in relation to these specifications.

A LIS is in direct conformance to these specifications when:

1. All datatypes are defined and mapped.
2. The value spaces of the healthcare datatypes used by the entity to be identical to the value spaces specified herein.
3. To the extent that the entity provides operations other than movement or translation of values, define operations on the healthcare datatypes which can be derived from, or are otherwise consistent with the characterizing operations specified herein.

Compliance

These specifications:

- Are in indirect conformance with ISO/DIS 21090/2008.
- Are in compliance with applicable sections of ISO 18308/2008.
- Are in compliance with applicable sections of ISO/TR 20514:2005.
- Are in compliance with applicable sections of ISO 13606-1:2007.
- Are in conformance with W3C XML Schema Definition Language (XSD) 1.1

Availability

The MLHIM specifications, reference implementation and tools are available from GitHub:

https://github.com/mlhim

Final versioned releases are available, packaged as .ZIP files from Launchpad:

http://launchpad.net/mlhim

Governance

The MLHIM project was originally started by four individuals with a desire to build an openly available approach to true syntactic and semantic interoperability for global healthcare IT. The founding agreement signed by these four guarantees that the MLHIM specifications documents and the reference implementation will be published under open source and open content licenses.

The MLHIM project is a meritocracy. Anyone may participate in the community which uses the Google Plus social network community for discussions.

Introduction

The Multi-Level Health Information Modeling (MLHIM) specifications are partially derived from some ISO Healthcare Information Standards (See the 'Compliance Secition') and the openEHR 1.0.2 specifications and the intent is that MLHIM 1.x be technologically inter-operable with openEHR.

MLHIM 2.x (this document and related artifacts) introduces further innovation through the use of XML technologies and reducing complexity without sacrificing interoperability as well as improved modeling tools and as application development platforms. These specifications can be implemented in any structured language. While a certain level of knowledge is assumed, the primary goal of these specifications is to make them 'implementable' by the widest possible number of people. The primary motivator for these specifications is the complexity involved in the recording of the temporal-spatial-ontological relationships in healthcare information while maintaining the original semantics across all applications; for all time.

We invite you to join us in this effort to maintain the specifications and build great, translatable healthcare tools and applications for global use.

International input is encouraged in order for the MLHIM specifications to allow for true interoperability, available to everyone in all languages and most of all, implementable by mere mortals.

Actual implementation in languages other than XML Schema and related XML technologies, the packages/classes should be implemented per the standard language naming format. A Language Implementation Specification (LIS) should be created for each language. For example MLHIM-Python-LIS.odt or MLHIM-Java-LIS.odt.

MLHIM intentionally does not specify full behavior within a class. Only the data and constraints are specified. Behavior may differ between various applications and should not be specified at the information model level. The goal is to provide a system that can capture and share the semantics and structure of information in the context in which it is captured.

The generic class names in the specification documents are in CamelCase type. since this is most typical of implementation usage.

Only the reference model is implemented in software where needed. In many implementations, the application can use XML Tools to validate against the CCD and reference model using a NoSQL storage. The domain knowledge models are implemented in the XML Schema language and they represent constraints on the reference model. These domain knowledge models are called Concept Constraint Definitions and the acronyms CCD and CCDs are used throughout MLHIM documents to mean these XML Schema files. This means that, since CCDs form a model that allows the creation of data instances from and according to a specific CCD, it is ensured that the data instances will be valid, in perpetuity.

However, any data instance should be able to be imported into any MLHIM based application since the root data model, for any application is the reference model. But, the full semantics of that data will not be known unless and until the defining CCD is available to the receiving application. The CCD represents the structural syntax of a healthcare[1] concept using XML Schema constraints and contains the semantics defined by the modeller in the form of RDF or other XML based syntaxes within documentation and annotation segments of the CCD. This enables applications to parse the CCD and publish or compute using the semantics as needed on an application by application basis.

The above paragraph describes the foundation of semantic interoperability in MLHIM implementations. You must understand this and the implications it carries to be successful with implementing MLHIM based applications. See the Constraint section for further discussion of Concept Constraint Definitions (CCDs).

1 Healthcare concepts include: (a) clinical concepts adopted in medicine for diagnostic and therapeutic decision making; (b) analogous concepts used for all healthcare professionals (e.g. nursing, dentistry, psychology, pharmacy); (c) public health concepts (e.g. epidemiology, social medicine, biostatitics); (d) demographic and (e) administrative concepts that concern healthcare

The MLHIM Eco-System

It is important here to describe all of the components of the MLHIM conceptual eco-system in order for the reader to appreciate the scope of MLHIM and the importance of the governance policies.

At the base of the MLHIM eco-system is the Reference Model (RM). Though the reference implementation is in XML Schema format, in real world applications a chosen object oriented language will likely be used for implementations. Often, tools are available to automatically generate the reference model classes from the XML Schema. This is the basis for larger MLHIM compliant applications. We will later cover implementation options for smaller applications such as mHealth (apps for smartphones and tablets, as well as purpose specific devices such as a home blood pressure monitor).

The next level of the MLHIM hierarchy is the Concept Constraint Definition (CCD). The CCD is a set of constraints against the RM that narrow the valid data options to a point where they can represent a specific healthcare concept. The CCD is essentially an XML Schema that uses the RM complex types as base types. This is conceptually equivalent to inheritance in object oriented applications, represented in XML Schema.

Since a CCD (by definition) can only narrow the constraints of the RM, then any data instance that is compliant with a CCD is also compliant in any software application that implements the RM or is designed to validate against the RM. Even if the CCD is not available, an application can know how to display and even analyze certain information. For example, if a receiving application does not have a CCD for a given data instance it will be able to discern the CCD ID and RM version from the element name and attributes of the root element. It may or may not be able to retrieve the CCD from the xsi:schemaLocation attribute. If not, it will still be able to determine, based on the reference model version, information about the data by using the names of elements nested within an element with the prefix 'ccd:'. Because these element names are unique to certain RM complexTypes. IF there is a <magnitude> element and a DVCount-units element, that fragment is a DvCount.

This is not to imply that all CCDs must be publicly available. It is possible to maintain a set of CCDs within a certain political jurisdiction or within a certain professional sector. How and where these CCDs are maintained are outside the scope of these specifications. Developers proficient in XML technologies will understand how this fits into their application environment.

This is now the point where the MLHIM eco-system is in contrast to the top-down approach used by other multi-level modelling specifications. In the real world; we know that there can never be complete consensus across the healthcare spectrum of domains, cultures and languages; concerning the details of a specific concept. Therefore the concept of a “maximal data model”, though idealistically valid, is realistically unattainable. Participation in and observation of these attempts to build concensus has led to the development of the Cavalini-Cook Theory stating that: The probability of reaching consensus among biomedical experts tends to zero with the increase of; the number of concepts considered and the number of experts included in the consensus panel.

In MLHIM, participants at any level are encouraged to create domain knowledge models that fit their needs. The RM has very little semantic context in it to get in the way. This allows structures to be created as the modeler sees fit for purpose. There is no inherent idea of a specific application in the RM, such as an Electronic Health Record (EHR), Electronic Medical Record (EMR), etc, although the MLHIM specifications can also be adopted for the development of these types of applications. This provides an opening for small, purpose specific apps such as mobile or portable device software as well.

In MLHIM, there is room for thousands of CCDs to describe each healthcare concept, (e.g. blood pressure, body temperature, problem list, medication list, Glasgow Coma Scale, cost of a medical procedure, or any other healthcare phenomena) vs. a single, flat model implemented in software that must encompass all descriptions/uses/etc. This multiplicity of compatible domain knowledge models is achieved by the way CCDs are uniquely identified by a Version 4 Universal Unique Identifier (UUID)[2] prefixed with 'ccd-'. CCDs are built out of pluggable complexTypes (PcTs) so that modelers can use granular definitions to create any size application form needed. Modelers and developers can create systems that allow users to choose between a selection of CCDs to include at specific points, at run-time.

With MLHIM CCDs you can deliver your data with complete syntactic interoperability and as much semantic interoperability and information exchange as the modeler chose to include in the CCD.

The governance of CCDs is left to the modeller and/or publishing organization. There are very strict guidelines that define what constitutes a valid CCD, as seen above.

A Valid CCD Must:

- Be a valid XML Schema 1.1 schema as determined by widely available parser/validators such as Xerces[3] or Saxon[4]
- Consist of complexTypes that only use restriction of complexTypes from the associated reference model

2 http://en.wikipedia.org/wiki/Universally_unique_identifier

3 http://xerces.apache.org/xerces2-j/faq-xs.html

4 http://www.saxonica.com/documentation/schema-processing/

- Import the reference model schema from www.mlhim.org using the appropriately defined namespace. Example for release 2.4.4

  <xs:import schemaLocation='http://www.mlhim.org/xmlns/mlhim2/2_4_4/mlhim244.xsd' namespace='http://www.mlhim.org/xmlns/mlhim2/2_4_4'/>
- use Type 4 UUIDs for complexType names, with the prefix of, 'ct-'. Example:

  <xs:complexType name='ct-8c177dbd-c25e-4908-bffa-cdcb5c0e38e6' xml:lang='en-US'>

  the language attribute is optional.
- publish a global element for each complexType with the name defined using the same UUID as the complexType with the 'ct-' prefix replaced with 'el-'. Example:

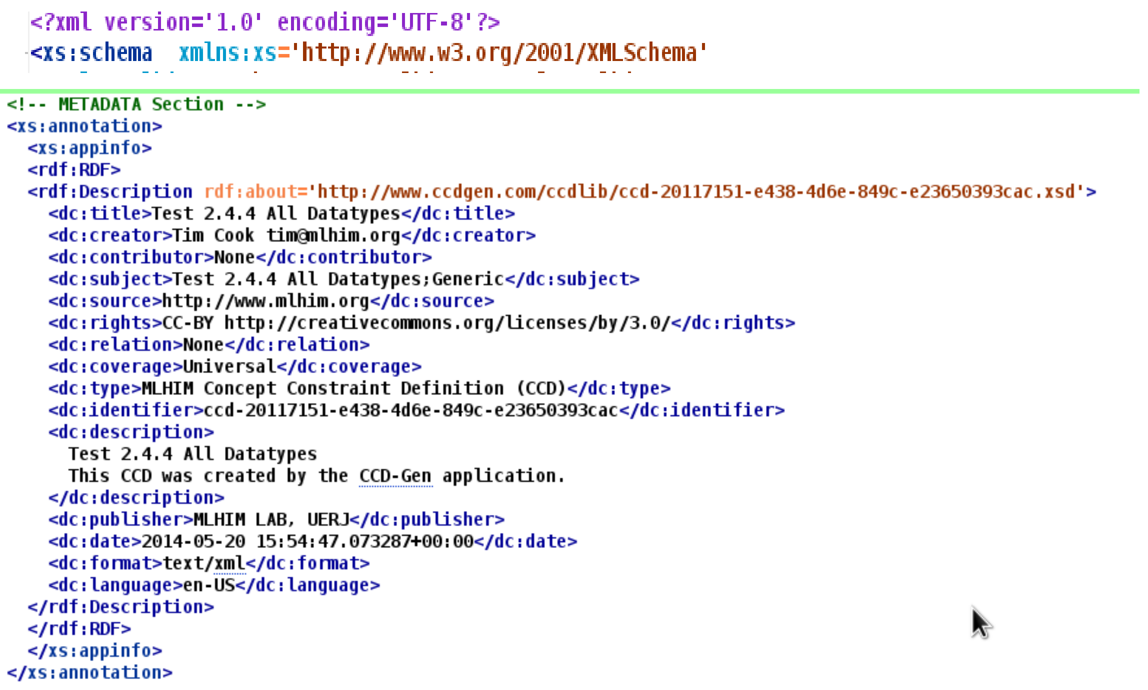
  <xs:element name='el-8c177dbd-c25e-4908-bffa-cdcb5c0e38e6' substitutionGroup='mlhim244:DvAdapter-dv' type='ccd:ct-8c177dbd-c25e-4908-bffa-cdcb5c0e38e6'/>
- use the correct substitution group(s) as in the example above
- define the namespaces in accordance with the namespaces in Table 1. An example from a CCD is shown in Figure 1.
- define the minimum DCMI[5] metadata items as shown in Figure 2.

prefix	namespace
mlhim244	http://www.mlhim.org/xmlns/mlhim2/2_4_4
ccd	http://www.mlhim.org/ccd
xs	http://www.w3.org/2001/XMLSchema
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
owl	http://www.w3.org/2002/07/owl#
dc	http://purl.org/dc/elements/1.1/
rdfs	http://www.w3.org/2000/01/rdf-schema#
targetNamespace	http://www.mlhim.org/ccd

Table 1: Namespaces Table.

Example schema fragment showing namespace definitions:

5 http://dublincore.org/



```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema  xmlns:xs='http://www.w3.org/2001/XMLSchema'
```

A

```
<!-- METADATA Section -->
<xs:annotation>
  <xs:appinfo>
  <rdf:RDF>
  <rdf:Description rdf:about='http://www.ccdgen.com/ccdlib/ccd-20117151-e438-4d6e-849c-e23650393cac.xsd'>
    <dc:title>Test 2.4.4 All Datatypes</dc:title>
    <dc:creator>Tim Cook tim@mlhim.org</dc:creator>
    <dc:contributor>None</dc:contributor>
    <dc:subject>Test 2.4.4 All Datatypes;Generic</dc:subject>
    <dc:source>http://www.mlhim.org</dc:source>
    <dc:rights>CC-BY http://creativecommons.org/licenses/by/3.0/</dc:rights>
    <dc:relation>None</dc:relation>
    <dc:coverage>Universal</dc:coverage>
    <dc:type>MLHIM Concept Constraint Definition (CCD)</dc:type>
    <dc:identifier>ccd-20117151-e438-4d6e-849c-e23650393cac</dc:identifier>
    <dc:description>
      Test 2.4.4 All Datatypes
      This CCD was created by the CCD-Gen application.
    </dc:description>
    <dc:publisher>MLHIM LAB, UERJ</dc:publisher>
    <dc:date>2014-05-20 15:54:47.073287+00:00</dc:date>
    <dc:format>text/xml</dc:format>
    <dc:language>en-US</dc:language>
  </rdf:Description>
  </rdf:RDF>
  </xs:appinfo>
</xs:annotation>
```

Figure 2: Minimum MetaData

Valid CCD Must Not:

- Contain any other language processing instructions required for validating instance data. For example; Schematron rules. While Schematron can be very valuable in some processing environments it is considered implementation specific and not part of the MLHIM interoperability framework.
- Import any XML Schema document other than its parent reference model schema.

MLHIM Modelling

Approach

The MLHIM specifications are arranged conceptually, into packages. These packages represent logical groupings of classes providing ease of consistent implementation. That said, the MLHIM Reference Model is implemented in a single XML Schema. The fundamental concepts, expressed in the reference model classes, are based on basic philosophical concepts of real world entities. These broad concepts can then be constrained to more specific concepts using models created by domain experts, in this case healthcare experts.

In MLHIM 1.x these constraints were known as archetypes, expressed in a domain specific language (DSL) called the archetype definition language (ADL). This language is based on a specific model called the archetype object model (AOM). MLHIM 1.x is a fork of the open source openEHR specifications.

In MLHIM 2.x and later, a given domain knowledge model is implemented in an XML Schema (XSD), called a Concept Constraint Definition (CCD). This allows MLHIM to use the XML Schema language as well as other XML technologies, as the constraint language. This provides the MLHIM development community with an approach to using multi-level modelling in healthcare using standardized, widely available tools and technologies.

The attempt to design a maximal data model for a concept is still restricted to the knowledge and context of a particular domain expert. In effect, there can never be a maximal data model for a healthcare concept. This means that from a global perspective there may be several CCDs that purport to fill the same need. There is no conflict in the MLHIM world in this case as CCDs are identified using the UUID and there are no semantics in the processing and validation model. The CCD may be further constrained at the implementation level through the use of implementation templates in the selected framework. Examples are additional XML Schemas that provide further restrictions or HTML pages used for display and data form entry. These templates are constructed in the implementation and may or may not be sharable across applications. The MLHIM specifications do not play any role in defining what these templates look like or perform like. They are only mentioned here as a way of making note that applications will require a user interface template layer to be functional.

The real advantage to using the MLHIM approach to healthcare information modelling is that it provides for a wide variety of healthcare applications to be developed based on the broad concepts defined in the reference model. By having domain experts within the healthcare field to define and create the CCDs, they can then be shared across multiple applications so that the structure and semantics of the data is not locked into one specific application, but can be exchanged among many different applications. This properly implements the separation of roles between IT and domain experts.

To demonstrate the differences between the MLHIM approach and the typical data model design approach we will use two common metaphors.

1. The first is for the data model approach to developing software. This is where a set of database definitions are created based on a requirements statement representing an information model. An application is then developed to support all of the functionality required to input, manipulate and output this data as information, all built around the data model. This approach is akin to a jigsaw puzzle (the software application) where the shape of the pieces are the syntax and the design and colors are the semantics, of the information represented in an aggregation of data components described by the model. This produces an application that, like the jigsaw puzzle, can provide for pieces (information) to be exchanged only between exact copies of the same puzzle. If you were to try to put pieces from one puzzle into a different puzzle, you might find that a piece has the same shape (syntax) but the picture on the piece (semantics) will not be the same. Even though they both belong to the same domain of jigsaw puzzles. You can see that getting a piece from one puzzle to correctly fit into another is going to require manipulation of the basic syntax (shape) and /or semantics (picture) of the piece. This can also be extended to the relationship that the puzzle has a defined limit of its four sides. It cannot, reasonably, be extended to incorporate new pieces (concepts) discovered after the initial design.

2. The multi-level approach used in MLHIM is akin to creating models (applications) using the popular toy blocks made by Lego® and other companies. If you compare a box of these interlocking blocks to the reference model and the instructions to creating a specific toy model (software application), where these instructions present a concept constraint (implemented as a CCD in MLHIM). You can see that the same blocks can be used to represent multiple toy models without any change to the physical shape, size or color of each block. Now we can see that when new concepts are created within healthcare, they can be represented as instructions for building a new toy model using the same fundamental building blocks that the original software applications were created upon.

Constraint Definitions

Concept Constraint Definitions (CCDs) can be created using any XML Schema or even a plain text editor. However, this is not a recommended approach. Realistic CCDs can be several hundred lines long. They also require Type4 UUIDs to be created as complexType and element names.

An open source Constraint Definition Designer (CDD) has been started but is in need of a leader and larger community. It is a tool to be used to create constraint definitions. It is open source and we hope to build a community around its development. The CDD can be used now to create a shell XSD with the correct metadata entries. Each release is available in the Tools section on the MLHIM GitHub site.

There is also a conceptual model of the information using the mind mapping software, Freemind. It provides domain experts a method of building up the structures to define a certain healthcare concept. There is a tool being developed to convert these Freemind definitions into a standardized format for import and processing by the CCD-Gen.

CCD Identification

The root element of a CCD and all complexType and global elements will use Type UUIDs as defined by the IETF RFC 4122 See: http://www.ietf.org/rfc/rfc4122.txt

The filename of a CCD may use any format defined by the CCD author. The CCD author must recognize that the metadata section of the CCD must contain the correct RDF:about URI with this filename. As a matter of consistency and to avoid any possible name clashes, the CCDs created by the CCD-Gen also use the CCD ID (ccd-<uuid>.xsd) To be a viable CCD for validation purposes the CCD should use the W3C assigned extension of '.xsd'. Though many tools may still process the artifact as an XML Schema without it.

The MLHIM community considers it a matter of good artifact management practice to use the CCD ID with the .xsd extension, as the filename.

CCD Versioning

Versioning of CCDs is not supported by these specifications. Though XML Schema 1.1 does have supporting concepts for versioning of schemas, this is not desirable in CCDs. The reasons for this decision focus primarily around the ability to capture temporal and ontological semantics. A key feature of MLHIM is the ability to guarantee the semantics for all future time, as intended by the original modeller. We determined that any change in the structure or semantics of a CCD, constitutes a new CCD. Since the complexTypes are re-usable (See the PcT description below), an approach that tools should use is to allow for copying a CCD and assigning a new CCD ID. When a complexType is changed within this new CCD, it also must be assigned a new name along with its global element name.

Pluggable complexTypes (PcTs)

MLHIM CCDs are made up of XML schema complexTypes composed by restriction of the Reference Model complexTypes. This is the foundation of interoperability. What is in the Reference Model is the superset of all CCDs. Pluggable complexTypes (PcTs) are a name we have given to the fact that due to their unique identification the complexTypes can be seen as re-usable components. For example, a domain expert might model a complexType that is a restriction of DvStringType with the enumerations for selecting one of the three measurement systems for temperature; Fahrenheit, Kelvin and Celsius. This PcT as well as many others can be reused in many CCDs without modification. For this reason, the semantic links for PcTs are directly expressed in an appinfo section in each PcT.

Implementations

It is the intent of the MLHIM community to maintain implementations and documentation in all major programming languages. Volunteers to manage these are welcome.

XML Schema

The reference implementation is expressed in XML Schema. Each release package contains the reference model schema as well as this and other documentation. The release and current development schemas live at the namespace on MLHIM.org

Best Practices

The following is an unordered list drawn from implementers questions and experience. The Best Practices are primarily focused on creating CCDs but they include some application development level suggestions as well.

How To Migrate a Legacy System

The first requirement is to have a good data dictionary of the existing system. Then convert each of these definitions into separate PcTs. The MLHIM Technological Development Unit has experience converting the United States National Cancer Institute's Common Data Element (NCI CDEs), the Agency for Healthcare Research and Quality (AHRQ)'s United States Health Information Knowledgebase (USHIK) Data Elements and the Fast Interoperability Healthcare Resources (FHIR) schema elements to discrete PcTs, but those are just a few examples. Actually, any data element required by any healthcare application or biomedical research database can be converted into PcTs, which can then be combined to build CCDs, for use in applications while remaining compliant with the structure and semantics of the legacy systems based on those example Data Elements. These PcTs are available on the CCD-Gen.

Other suggestions appreciated.

The Reference Model

Please note: Any discrepancies between this document and the XML Schema implementation is to be resolved by the XML Schema. The automatically generated XML Schema documentation is available in PDF and downloadable HTML forms: http://mlhim.org/documents.html The sources are maintained on GitHub at https://github.com/mlhim/specs To get the most recent development version using git create a clone of : https://github.com/mlhim/specs.git

Assumed Types

There are several types that are assumed to be supported by the underlying implementation technology. These assumed types are based on XML Schema 1.1 Part 2 Datatypes. They should be available in your implementation language or add-on libraries. The names may or may not be exactly the same.

Non-Ordered Types:

Name	Description
anyType	Also sometimes called Object. This is the base class of the implementation technology.
boolean	Two state only. Either true or false.
string	The string data type can contain characters, line feeds, carriage returns, and tab characters.
normalizedString	The normalized string data type also contains characters, but all line feeds, carriage returns, and tab characters are removed.
token	The token data type also contains characters, but the line feeds, carriage returns, tabs, leading and trailing spaces are removed, and multiple spaces are replaced with one space.
anyURI	Specifies a Unique Resource Identifier (URI).
hash	An enumeration of any type with a key:value combination. The keys must be unique.
list	An ordered or unordered list of any type.
set	An ordered or unordered but unique list of any type.

Table 2: Assumed: Non-ordered types

Name	Description

dateTime	The dateTime data type is used to specify a date and a time.
	The dateTime is specified in the following form "YYYY-MM-DDThh:mm:ss" where:
	YYYY indicates the year
	MM indicates the month
	DD indicates the day
	T indicates the start of the required time section
	hh indicates the hour
	mm indicates the minute
	ss indicates the second
	The following is an example of a dateTime declaration in a schema:
	`<xs:element name="startdate" type="xs:dateTime"/>`
	
	An element in your document might look like this:
	`<startdate>2002-05-30T09:00:00</startdate>`
	Or it might look like this:
	`<startdate>2002-05-30T09:30:10:05</startdate>`
	
	Time Zones
	To specify a time zone, you can either enter a dateTime in UTC time by adding a "Z" behind the time - like this:
	`<startdate>2002-05-30T09:30:10Z</startdate>`
	or you can specify an offset from the UTC time by adding a positive or negative time behind the time - like this:
	`<startdate>2002-05-30T09:30:10-06:00</startdate>`
	or
	`<startdate>2002-05-30T09:30:10+06:00</startdate>`

date	The date data type is used to specify a date. The date is specified in the following form "YYYY-MM-DD" where: YYYY indicates the year MM indicates the month

Name	Description
	DD indicates the day An element in an XML Document might look like this: <start>2002-09-24</start>
time	The time data type is used to specify a time. The time is specified in the following form "hh:mm:ss" where: hh indicates the hour mm indicates the minute ss indicates the second The following is an example of a time declaration in a schema: <xs:element name="start" type="xs:time"/> An element in your document might look like this: <start>09:00:00</start> Or it might look like this: <start>09:30:10:05</start> **Time Zones** To specify a time zone, you can either enter a time in UTC time by adding a "Z" behind the time - like this: <start>09:30:10Z</start> or you can specify an offset from the UTC time by adding a positive or negative time behind the time - like this: <start>09:30:10-06:00</start> or <start>09:30:10+06:00</start>
duration	The duration data type is used to specify a time interval. The time interval is specified in the following form "PnYnM-nDTnHnMnS" where: P indicates the period (required) nY indicates the number of years nM indicates the number of months nD indicates the number of days T indicates the start of a time section (required if you are going to

Name	Description

	specify hours, minutes, or seconds)
	nH indicates the number of hours
	nM indicates the number of minutes
	nS indicates the number of seconds
	The following is an example of a duration declaration in a schema:
	`<xs:element name="period" type="xs:duration"/>`
	
	An element in your document might look like this:
	`<period>P5Y</period>`
	The example above indicates a period of five years.
	Or it might look like this:
	`<period>P5Y2M10D</period>`
	The example above indicates a period of five years, two months, and 10 days.
	Or it might look like this:
	`<period>P5Y2M10DT15H</period>`
	The example above indicates a period of five years, two months, 10 days, and 15 hours.
	Or it might look like this:
	`<period>PT15H</period>`
	The example above indicates a period of 15 hours.
	Negative Duration
	To specify a negative duration, enter a minus sign before the P:
	`<period>-P10D</period>`
	The example above indicates a period of minus 10 days.

Partial Date Types	Support for partial dates is essential to avoid poor data quality. In order to provide for partial dates and times the following types are assumed to be available in the language or in a library. Day – provide on the day of the month, 1 – 31 Month – provide only the month of the year, 1 – 12 Year – provide on the year, CCYY MonthDay – provide only the Month and the Day (no year) YearMonth – provide only the Year and the Month (no day)

Name	Description
real	The decimal data type is used to specify a numeric value. Note: The maximum number of decimal digits you can specify is 18.
integer	The integer data type is used to specify a numeric value without a fractional component.

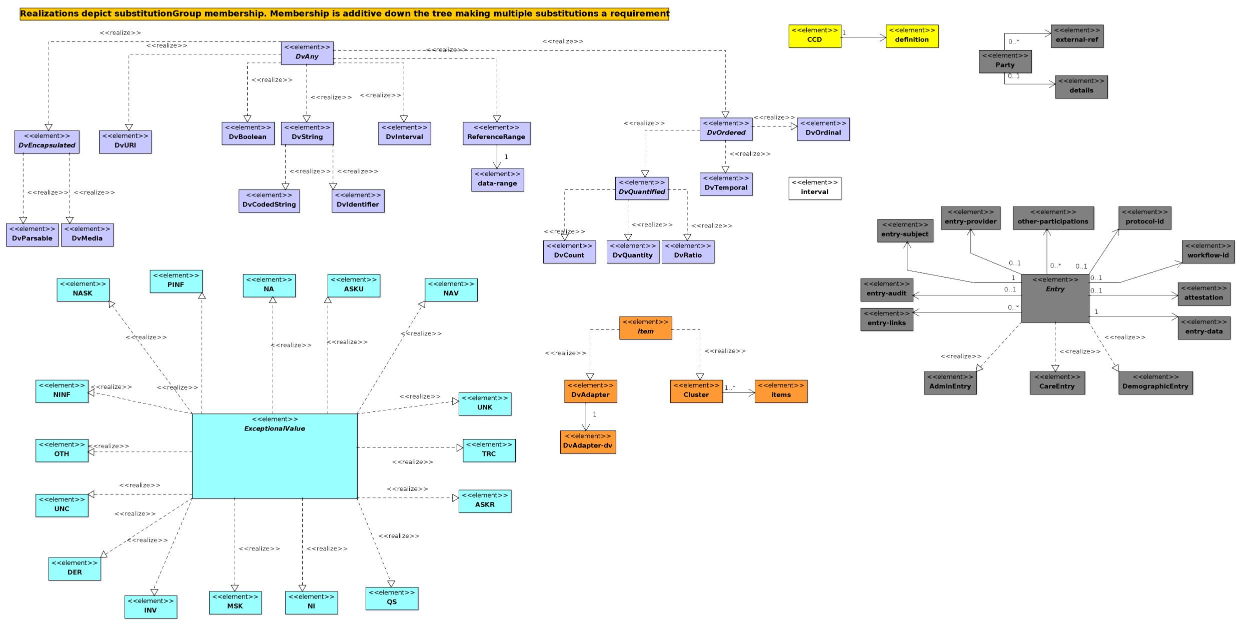
Table 3: Assumed: Ordered Types

Technical Documentation

2.4.4 Reference Model

The complete documentation in a graphical, clickable format is available on the MLHIM web-site via this Reference implementation link.

Drawing 1 depicts the global elements of the reference implementation. This full size image as well as the ArgoUML project is avaible in the UML directory of the specifications.



Drawing 1: Global Elements

The reference implementation complexType descriptions.

DataValue Group

DvAnyType

Derived from: n/a

Abstract: True

Description: Serves as a common ancestor of all data-types in MLHIM models.

DvBooleanType

Derived from: DvAnyType **by extension**

Abstract: False

Description: Items which represent boolean decisions, such as true/false or yes/no answers. Use for such data, it is important to devise the meanings (usually questions in subjective data) carefully, so that the only allowed results are in fact true or false.

Potential Misuse: The DvBooleanType should not be used as a replacement for naively modelled enumerated types such as male/female etc. Such values should be coded, and in any case the enumeration often has more than two values. Though the DvBoolean-dv element is a String type this is to easily allow responses that the user is more familiar with using in the context such as 'Yes', 'No' or 'True', 'False'. A conversion method is required to convert the valid-trues to True and the valid-falses to False.

DvURIType

Derived from: DvAnyType **by extension**

Abstract: False

Description: Used to specify a Universal Resource Identifier. Set the pattern to accommodate your needs in a CCD.

DvStringType

Derived from: DvAnyType **by extension**

Abstract: False

Description: The string data type can contain characters, line feeds, carriage returns, and tab characters.

DvCodedStringType

Derived from: DvStringType **by extension**

Abstract: False

Description: A text item whose DvString-dv element must be the long name or description from a controlled terminology. The key (i.e. the 'code') of which is the terminology-code attribute. In some cases, DvString-dv and terminology-code may have the same content.

DvIdentifierType

Derived from: DvStringType **by extension**

Abstract: False

Description: Type for representing identifiers of real-world entities. Typical identifiers include: drivers license number, social security number, veterans affairs number, prescription id, order id, system id and so on. The actual identifier is in the DvString-dv element.

DvEncapsulatedType

Derived from: DvAnyType **by extension**

Abstract: True

Description: Abstract class defining the common meta-data of all types of encapsulated data.

DvParsableType

Derived from: DvEncapsulatedType **by extension**

Abstract: False

Description: Encapsulated data expressed as a parsable String. The internal model of the data item is not described in the MLHIM model, but in this case, the form of the data is assumed to be plain-text, rather than compressed or other types of large binary data. If the content is to be binary data then use a DvMediaType.

DvMediaType

Derived from: DvEncapsulatedType **by extension**

Abstract: False

Description: A specialization of DvEncapsulatedType for audiovisual and bio-signal types. Includes further metadata relating to media types which are not applicable to other subtypes of DvEncapsulated.

DvOrderedType

Derived from: DvAnyType **by extension**

Abstract: True

Description: Abstract class defining the concept of ordered values, which includes ordinals as well as true quantities. The implementations require the functions '<', '>' and is_strictly_comparable_to ('==').

DvOrdinalType

Derived from: DvOrderedType **by extension**

Abstract: False

Description: Models rankings and scores, e.g. pain, Apgar values, etc, where there is

a) implied ordering,

b) no implication that the distance between each value is constant, and

c) the total number of values is finite.

Note that although the term ‘ordinal’ in mathematics means natural numbers only, here any decimal is allowed, since negative and zero values are often used by medical professionals for values around a neutral point. Also, decimal values are sometimes used such as 0.5 or .25

Examples of sets of ordinal values:

- -3, -2, -1, 0, 1, 2, 3 -- reflex response values
- 0, 1, 2 -- Apgar values

Used for recording any clinical datum which is customarily recorded using symbolic values.

Example: the results on a urinalysis strip, e.g. {neg, trace, +, ++, +++} are used for leukocytes, protein, nitrites etc; for non-haemolysed blood {neg, trace, moderate}; for haemolysed blood {neg, trace, small, moderate, large}

DvQuantifiedType

Derived from: DvOrderedType **by extension**

Abstract: True

Description: Abstract type defining the concept of true quantified values, i.e. values which are not only ordered, but which have a precise magnitude.

DvCountType

Derived from: DvQuantifiedType **by extension**

Abstract: False

Description: Countable quantities. Used for countable types such as pregnancies and steps (taken by a physiotherapy patient), number of cigarettes smoked in a day, etc. Misuse:Not used for amounts of physical entities (which all have standardized units). Note that PcTs derived from DvCountType should make magnitude, error and accuracy attributes minOccurs = '1'.

DvQuantityType

Derived from: DvQuantifiedType **by extension**

Abstract: False

Description: Quantified type representing “scientific” quantities, i.e. quantities expressed as a magnitude and units. Can also be used for time durations, where it is more convenient to treat these as simply a number of individual seconds, minutes, hours, days, months, years, etc. when no temporal calculation is to be performed. Note that PcTs derived from DvQuantityType should make magnitude, error and accuracy attributes minOccurs = '1'.

DvRatioType

Derived from: DvQuantifiedType **by extension**

Abstract: False

Description: Models a ratio of values, i.e. where the numerator and denominator are both pure numbers. Should not be used to represent things like blood pressure which are often written using a ‘/’ character, giving the misleading impression that the item is a ratio, when in fact it is a structured value. Similarly, visual acuity, often written as (e.g.) “6/24” in clinical notes is not a ratio but an ordinal (which includes non-numeric symbols like CF = count fingers etc). Should not be used for formulations.

DvTemporalType

Derived from: DvOrderedType **by extension**

Abstract: False

Description: Type defining the concept of date and time types. Must be constrained in PcTs to be one or more of the below elements. This gives the modeller the ability to optionally allow full or partial dates at run time. Setting maxOccurs and minOccurs to zero causes the element to be prohibited.

DvIntervalType

Derived from: DvAnyType **by extension**

Abstract: False

Description: Generic type defining an interval (i.e. range) of a comparable type. An interval is a contiguous subrange of a comparable base type. Used to define intervals of dates, times, quantities, etc. Whose datatypes are the same and are ordered. Implementations are tasked with coercing the string values in 'upper' and 'lower' into the type defined by interval-type.

ReferenceRangeType

Derived from: DvAnyType **by extension**

Abstract: False

Description: Defines a named range to be associated with any ORDERED datum. Each such range is particular to the patient and context, e.g. sex, age, and any other factor which affects ranges. May be used to represent normal, therapeutic, dangerous, critical etc ranges.

Common Group

FeederAuditDetailsType

Derived from: n/a

Abstract: False

Description: Audit details for any system in a feeder system chain. Audit details here means the general notion of who/where/when the information item to which the audit is attached was created. None of the elements are defined as mandatory, however, in different scenarios, various combinations of elements will usually be mandatory. This can be controlled by specifying feeder audit details in PcTs used when conjunction with non-MLHIM systems as interface definitions.

FeederAuditType

Derived from: n/a

Abstract: False

Description: Audit and other meta-data for software applications and systems in the feeder chain. This information is not typically used by modellers but by the applications themselves to "tag" entries when performing an extract.

PartyType

Derived from: n/a

Abstract: False

Description: Description of a party, including an optional external link to data for this party in a demographic or other identity management system. An additional details element provides for the inclusion of information related to this party directly. If the party information is to be anonymous then do not include the details element. The string 'Self' may be entered as the party-name if an external_ref is include.

AttestationType

Derived from: n/a

Abstract: False

Description: Record an attestation by a party of item(s) of record content. The type of attestation is recorded by the reason attribute, which my be coded.

ParticipationType

Derived from: n/a

Abstract: False

Description: Model of a participation of a Party (any Actor or Role) in an activity. Used to represent any participation of a Party in some activity, which is not explicitly in the model, e.g. assisting nurse. Can be used to record past or future participations. Should not be used in place of more permanent relationships between demographic entities.

ExceptionalValueType

Derived from: n/a

Abstract: True

Description: Subtypes are used to indicate why a value is missing (Null) or is outside a measurable range. The element ev-name is fixed in restricted types to a descriptive string. The subtypes defined in the reference model are considered sufficiently generic to be useful in many instances. CCDs may contain additional ExceptionalValueType restrictions.

NIType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *No Information* : The value is exceptional (missing, omitted, incomplete, improper). No information as to the reason for being an exceptional value is provided. This is the most general exceptional value. It is also the default exceptional value.

MSKType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Masked* : There is information on this item available but it has not been provided by the sender due to security, privacy or other reasons. There may be an alternate mechanism for gaining access to this information.

Warning:

Using this exceptional value does provide information that may be a breach of confidentiality, even though no detail data is provided. Its primary purpose is for those circumstances where it is necessary to inform the receiver that the information does exist without providing any detail.

INVType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Invalid* : The value as represented in the instance is not a member of the set of permitted data values in the constrained value domain of a variable.

DERType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Derived* : An actual value may exist, but it must be derived from the provided information; usually an expression is provided directly.

UNCType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Unencoded* : No attempt has been made to encode the information correctly but the raw source information is represented, usually in free text.

OTHType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Other*: The actual value is not a member of the permitted data values in the variable. (e.g., when the value of the variable is not by the coding system)

NINFType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Negative Infinity* : Negative infinity of numbers

PINFType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Positive Infinity* : Positive infinity of numbers

UNKType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Unknown* : A proper value is applicable, but not known.

ASKRType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Asked and Refused* : Information was sought but refused to be provided (e.g., patient was asked but refused to answer)

NASKType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Not Asked* : This information has not been sought (e.g., patient was not asked)

QSType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Sufficient Quantit*y : The specific quantity is not known, but is known to non-zero and it is not specified because it makes up the bulk of the material; Add 10mg of ingredient X, 50mg of ingredient Y and sufficient quantity of water to 100mL.

TRCType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Trace* : The content is greater or less than zero but too small to be quantified.

ASKUType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Asked but Unknown* : Information was sought but not found (e.g., patient was asked but did not know)

NAVType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Not Available* : This information is not available and the specific reason is not known.

NAType

Derived from: ExceptionalValueType **by restriction**

Abstract: False

Description: *Not Applicable* : No proper value is applicable in this context e.g.,the number of cigarettes smoked per day by a non-smoker subject.

Items Group

ItemType

Derived from: n/a

Abstract: True

Description: The abstract parent of ClusterType and DvAdapterType structural representation types.

ClusterType

Derived from: ItemType **by extension**

Abstract: False

Description: The grouping variant of Item, which may contain further instances of Item, in an ordered list. This provides the root Item for arbitrarily complex structures.

DvAdapterType

Derived from: ItemType **by extension**

Abstract: False

Description: The leaf variant of Item, to which any DvAnyType subtype instance is attached for use in a Cluster.

Entry Group

EntryType

Derived from: n/a

Abstract: True

Description: The abstract parent of all Entry subtypes. An Entry is the root of a logical set of data items. Each subtype has an identical information structure. The subtyping is used to allow persistence to separate the types of Entries; primarily import in healthcare for the de-identification of clinical information.

CareEntryType

Derived from: EntryType **by extension**

Abstract: False

Description: Entry subtype for all entries related to care of an a subject of record.

AdminEntryType

Derived from: EntryType **by extension**

Abstract: False

Description: Entry subtype for administrative information, i.e. information about setting up the clinical process, but not itself clinically relevant. Archetypes will define contained information. Used for administrative details of admission, episode, ward location, discharge, appointment (if not stored in a practice management or appointments system). Not used for any clinically significant information.

DemographicEntryType

Derived from: EntryType **by extension**

Abstract: False

Description: Entry subtype for demographic information, i.e. name structures, roles, locations, etc. Modelled as a separate type from AdminEntry in order to facilitate the separation of clinical and non-clical information to support de-identification of clinical and administrative data.

Constraint Group

CCDType

Derived from: n/a

Abstract: False

Description: This is the root node of a Concept Constraint Definition.

Example CCD

Please check the website documents section as well as the CCD Library on the CCD-Gen. The CCD-Gen requires free registration in order to view the CCD Library.

Concept Constraint Definition Generator (CCD-Gen)

Creating and Managing CCDs

The CCD-Gen is an online tool used to create CCDs via a web driven, declarative environment. The CCD-Gen allows the building of complete CCDs by selecting the desired Pluggable complex-Type (PcT) definitions from existing PcTs or ones that you design yourself. You assemble the pieces into the concept definition you need. By re-using PcTs you improve the data exchange and analysis capability. Many of the PcTs have been designed based on existing data models. Resources such as the Common Data Element definitions from the US National Cancer Institute, the United States Health Information Knowledgebase (USHIK) from the Agency for Healthcare Research and Quality (AHRQ) and HL7 FHIR® models have been translated to PcTs. More are planned and you can contribute to the open content effort.