DDI-CDI Syntax Representation and Modelling Notes

# Overview and Purpose

This document describes the modelling approach and techniques used to produce syntax-specific representations of the model used for the Data Documentation Initiative – Cross Domain Integration (DDI-CDI) specification. This is not the full documentation for each step of the production flows, which is covered by separate documentation, but it does describe the process and the steps employed, and the reasons why these approaches were taken.

This document makes reference to the UML Class Model Interoperable Subset (UCMIS). Although UCMIS was originally conceived as a tool for use within the DDI-CDI project, it has subsequently been recognized that it could be used also for other standards. Because of this, it has been documented separately, and could itself become a DDI work product in time. For the specifics, see the UCMIS description at <https://htmlpreview.github.io/?https://bitbucket.org/ddi-alliance/ucmis/raw/master/build/index.html>.

Models typically undergo many revisions, and can continue to change almost up to the publication date. Model-driven generation of documentation and syntax representations can immediately reflect the current status of the model, making these changes much easier to manage. The consistency of the model and the syntax representations are crucial, and a model-driven approach enables high quality in this regard. In a model-driven approach, the model is the “one source of truth,” even though it may be interpreted to be appropriate for diverse outputs. In the Object Management Group’s (OMG) model-driven architecture on the Meta Object Framework (MOF) at <https://www.omg.org/mda/executive_overview.htm> we read: “This model remains stable as technology evolves, extending and thereby maximizing software ROI.” This is basic intent behind the approach taken with the DDI-CDI model-driven approach.

In this document, we will look at the production flows from the model to the different outputs, consider how these relate to the UCMIS, examine the syntax representations and mapping to RDF and XML, and then provide some comments on the modelling conventions employed in the DDI-CDI model itself.

# I. Production Flow

The transformation of the DDI-CDI model into the different outputs takes several steps, and it is necessary to understand these steps in order to understand the various mappings which are used. The DDI-CDI Working Group (DDI-CDI WG) employs Enterprise Architect (EA) as its primary modelling application, but the specification itself – including the UML model – must be available for use in other modelling tools as well, since it is the core of the specification. The description of production flows starts with the internal files of the DDI-CDI WG and goes on to describe all the subsequent steps, as these are transformed in stages into the final product, and there are dependencies between them.

1. Start with EA model, realized as formal definitions plus some hand-crafted diagrams to illustrate core concepts (all in EA file).

2. Export the model into EA-proprietary XMI (only formal definitions, not the EA-generated diagrams) – images are captured as separate image files. Note that the XMI does not contain the “development” folder from the EA file (this is not published).

3. EA XMI is transformed by XSLTs into Canonical XMI. The output of the XSLTs is designed such that other proprietary XMI formats are supported (those used by Rational Rose, Eclipse, etc.).

4. Canonical XMI is produced in 3 versions, all contain the complete model:

(1) No unique association names within the package (they are short and more user-friendly but some tools expect these to be unique to the containing package)

(2) Unique association names within the package (created by adding the names of classes at either end) - good for import into any tool except Eclipse

(3) Eclipse version of the XMI w/ package-unique association names and different namespaces for UML and UML primitives

5. The Eclipse XMI is imported into the Eclipse Modelling Framework as the basis of syntax generation.

6. Acceleo functions within the Eclipse Modelling Framework, implementing the Model2Text specification from OMG for doing transformations.

“UCMIS Model2Text” is a tool we have written to produce syntax representations from UCMIS-compliant models. For this, see the repository (including description): <https://bitbucket.org/wackerow/ucmis.m2t/>.

UCMIS.M2T is a tool for the generation of the classifier documentation and syntax representations of a model confirming to the UML Class Model Interoperable Subset (UCMIS) using the Eclipse Acceleo (<https://www.eclipse.org/acceleo/>) implementation of the OMG standard MOF Model to Text Transformation Language (<https://www.omg.org/spec/MOFM2T/>).

7. Field-level documentation is generated (uses the XMI fragments from (2)). This produces a page per class and a page per datatype, produces lists of all related items, and generates SVG diagrams (not hand-crafted) which can be reused. Each page (per class, per datatype) has a URL generated based on the unique name of the class/datatype.

8. The XML XSD Schema is generated.

9. Two RDF Expressions are generated: Ontology as OWL and RDFA-S in Turtle syntax and in JSON-LD.

These transformations are shown in the following flowchart:



# II. UCMIS

UCMIS was developed because the Canonical XMI standard as published by W3C (<https://www.w3.org/TR/xml-c14n/>) is by itself insufficient to guarantee support by a sufficient range of UML modelling tools. Sections III, IV, and V of the DDI-CDI Public Review draft provide details on the Canonical XMI (Part\_3\_DDI-CDI\_Architecture\_PR\_1.pdf). UCMIS was developed by identifying what features of UML were both expressed in Canonical XMI and were also supported by the widest possible range of UML tools, a process which primarily involved field testing of the various applications.

UCMIS is essentially a subset of the class diagram feature in UML. It has several characteristics, summarized in the bullets below. Ability to map to OO languages on the conceptual level

* + Most common items from object-oriented (OO) data world/class diagrams
	+ Only single inheritance so supportable for more targets (mapping)
	+ Bi-directional (binary) associations (as opposed to associations with 3 ends) to support more mapping targets

There is a description of each of the supported items available in the Part\_3\_DDI-CDI\_Architecture\_PR\_1.pdf of the public review cited above. The UCMIS repository can be found at at <https://bitbucket.org/ddi-alliance/ucmis/>. This provides a description (including some short wording on UML) of UCMIS at <https://htmlpreview.github.io/?https://bitbucket.org/ddi-alliance/ucmis/raw/master/build/index.html>. There is also a detailed spreadsheet of UCMIS items at <https://htmlpreview.github.io/?https://bitbucket.org/ddi-alliance/ucmis/raw/master/build/ucmis_table.html>.

# III. Mapping to Target Syntaxes

Mappings to target syntax representations is primarily a mapping between the features of UCMIS and the features of the target syntax, guaranteeing a consistency of approach. It is the situation that specific instances of UCMIS (DDI-CDI) may present special cases, but these are intended to be as limited as possible.

Mapping is based on the generic object-oriented approach common to UCMIS and many object-oriented (OO) languages (the data modelling parts – not methods, etc.): class definitions and attributes, generalized and specialized classes, and associations. Exceptions must be well-documented.

The characteristics of these mappings can be summarized as follows:

* Class definitions and associations correspond 1-to-1
* Class attributes defined by data types
* Data types comprehend UML primitives, structured data types, and enumerations

\*\*NOTE that UML is not aware of XML datatypes: we have made UML primitives based on the small number of these which we use – a direct correspondence)\*\*

In mapping to any target language, all of the above features are mapped as far as possible. If something cannot be directly mapped, an exception is made. Again, these should be indicated.

When we map associations to any other form, it is important to do this in a consistent fashion. In UML, attributes and associations are very similar: often, the technique for mapping attributes serves as the basis for mapping associations, as this reflect the style of UML itself.

Associations in the CDI model use simple associations, aggregations, and compositions. The latter two are only used for documentation purposes, and are ignored in any mapping to a syntax. (There is no such distinction in most OO languages).

Note that the UML “XML Schema data types” are by definition mapped against actual XML Schema data types for all syntax representations (they all recognize these, unlike UML). The XML Schema data types exist most target languages, or have existing mappings/libraries.

The five UML primitives which are pre-defined are mapped to corresponding XML Schema data types.

XML Schema:

This syntax representation is meant to be a consistent serialization format, rather than a document-oriented hierarchical depiction of the model. Containership is only on the class level, and is not used for creating more typical XML deep hierarchies. (This resulted from the input of the DDI developer community during the DDI 4 evolution.) The graph-oriented structure of the UML model is reflected in the XML Schema with references.

The XML schema is basically a 1-to-1 representation of the model.

* Classes and super-classes are realized as complex types (super-classes are extension bases for sub-classes)
* Associations are expressed as child-element containers for the references to the associated class. These child elements are named after the association predicates. Example: The AgentListing element has a child element “AgentListingIsDefinedByConcept” which has a type of ReferenceXSDType. (While verbose, this avoids the problems related to the element container of reused elements.)
* Class attributes are child elements, typed according to the related primitives, structured data types, or enumerations.
* Structured data types are realized as complexTypes in the same way as classes – these have no associations. Note that in the XML, there is no visible distinction between the classes in the model and the structured data types.

RDF:

In the RDF mapping, the correspondences are quite direct:

* UML Classes are RDF-S classes and OWL classes at the same time
* UML Class attributes are RDF properties and OWL object properties at the same time
* Attributes with a data type in UML are defined in RDF with an RDF-S range which references a structured data type
* Structured data types are RDF-S and OWL classes at the same time
* Enumerations are also RDF-S classes and OWL classes at the same time; the enumeration literals are an instance of the specific enumeration (Example: in the ControlConstruct enumeration, one literal is “Else” – this becomes an instance of the enumeration ControlConstruct, so that in the corresponding instance graph there is an RDF literal “Else”. )
* As in UML, identifiers (agency – id – version) are expressed as attributes of a structured data type (that is, as properties). This set of information is described in OWL as a compound key. For example, a reasoner can understand that these properties form a key.

JSON-LD Differences:

In the JSON-LD, enumerations don’t exist because there is no corresponding construct in JSON-LD.

Again, as with the XML Schema representation, the XML Schema data types are used directly. The predefined UML primitives and the created UML primitives for the used XML Schema data types are mapped to the XML Schema data types.

The details of the mapping in all mentioned cases can be seen in the documentation page per class/datatype. Each page shows a the related fragment of a class/datatype in the target language.

# IV. Stylistic Conventions – Modelling DDI-CDI

**Design Patterns**

There are a set of design patterns in the model which exist for the purposes of guaranteeing consistency in the model, and for developers who wish to understand the places where model constructs will be consistent. These design patterns ore strictly informational, and are not reflected in the syntax representations other than as implemented in the model itself.

**Associations**

Associations are subject – predicate – object (the concept of target and source is an EA convention which has caused much confusion, as it is not an actual UML construct.)

**Identifiers**

DDI-CDI has identifiers per class (this is not required by UCMIS). Identifiers in UML are just garden-variety class attributes. This is reflected in the XML Schema (they are just child elements like any other class attribute. Identifiers are a structured data type which reflects the ISO-11179-based traditional tri-partite structure of DDI: agency – id – version. There is no URN field – we always break it out into three parts for the formal DDI-CDI identifier. You can always assemble a DDI URN from this information. Note that we do support URNs for referencing, so that we can make reference to things in DDI Lifecycle, using the URI attribute which can also hold URLs for external things.)

**References**

The references in DDI-CDI are consistently implemented with a complex data type designed for this purpose. While not identical in detail, this construct is similar to and based on the DDI Lifecycle type which performs a similar purpose. This is an area where some implementations of the model may diverge from the model, especially given that RDF representations use URIs for all references (indeed, for all relationships of any kind) and individual communities of users may choose to use a syntax-native form of referencing in preference to the mechanism used in the model. The “official” DDI syntax representations do not do this, but it is to be expected within specific communities implementations.

**Sequences and Ordering**

The model does not rely on implicit sequences as expressed in any particular syntax representation. In those cases where sequence or order are important, properties indicating position in the sequence are explicit. This is a necessary aspect of the syntax neutrality of the model, as different target syntaxes may handle sequences in different ways. In the “official” syntax representations, ordering is always indicated explicitly (e.g., no relying on the implicit ordering within the XML expression of the model.)

**From the Public Review Architecture Document**

For convenience, the following material is copied from the Public Review Architecture document, as it reflects more information relevant to the modelling style of DDI-CDI.

## Notes on Modeling

### Structural Items

#### Package

* A package expresses a region of the interrelated content
* Packages (and classes) are named and organized in a way that they can be easily moved to another location of the model
	+ The name of each package is a unique name (in the scope of all items) in the whole model

#### Class

A class can be understood as a blueprint for an object. It describes the type of objects for which DDI-CDI is a model for.

* Classes (and packages) are named and organized in a way that they can be easily moved to another location of the model
	+ The name of each class must be a unique name (in the scope of all items) in the whole model
* Attributes are used
* Many classes have the attributes agency, id, and version. These items build a composite identifier aligned with the international registration data identifier (IRDI)[[1]](#footnote-1). Instantiated objects of these classes can be globally uniquely identified by these identifiers.
This approach enables reuse of these objects on a granular level.

#### Attribute

A class attribute is typed by a data type.

### Relationships

#### Association

Only binary associations are used, i.e. two classes are related.

A notion of direction is used in DDI-CDI to be able to properly name associations so that they read as semantic triple (subject-predicate-object). Additionally, the direction is defined by a navigable association end at the “object” class. The association has unspecified navigability at the end of the “subject” class.

Associations are rendered in diagrams by connecting classes with a line. The navigable end is indicated by an open arrowhead (🡪) on one end of an association and owned by the class on the opposite end.

The definition of direction/navigation can be understood just as a recommendation, see also this citation from the official UML specification in the section on the semantics of associations.

*„Navigability means that instances participating in links at runtime (instances of an Association) can be accessed efficiently from instances at the other ends of the Association. The precise mechanism by which such efficient access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.*

*NOTE. Tools operating on UML models are not prevented from navigating Associations from non-navigable ends.”*[[2]](#footnote-2)

“*Specifying a direction of traversal does not necessarily mean that you can't ever get from objects at one end of an association to objects at the other end. Rather, navigation is a statement of efficiency of traversal.*”[[3]](#footnote-3)

For specific uses, the direction of an association might not make sense. In this case, the navigation definition can be just ignored.

Association names should be unique in one package if possible. This is not always suitable in terms of achieving short names. Some UML tools comply in a strict sense to the UML rule that elements of related or the same type should have unique names within the enclosing package. For this purpose, a second representation of the model in Canonical XMI is provided which has unique association names per package.

#### Multiplicity

Multiplicity is formally defined as a lower and upper bound. Simply put: a multiplicity is made up of a lower and an upper cardinality. Cardinality is how many elements are in a set.

The default multiplicity of the “subject” class is 0..n. The multiplicity of the “object” class is usually 0..1 or 0..n. Zero for the lower cardinality allows flexibility in the process of producing metadata.

**Association rendering:**



#### Aggregation

Any semantics in aggregation are not seen which are not already covered by a common association with appropriate directed names, but it could provide a way of easily visualizing a whole/part relationship.

**Aggregation rendering:**



#### Composition

Composition is used for cases in which there is a strong lifecycle dependency, e.g. a cell in an array cannot exist without the array. However, it could provide a way of easily visualizing strong lifecycle dependency.

**Composition rendering:**



#### Generalization

A class can be an extension of another class (the general class). Attributes and associations of the general class are inherited. Only single inheritance is used, i.e. a class can only extend one other.

A data type can be an extension of another data type (the general data type). Attributes of the general data type are inherited. Only single inheritance is used, i.e. a data type can only extend one other. This applies also to primitive data types and enumerations.

**Class generalization rendering:**



### Data Type Definition

#### Data Type

A data type can be a UML primitive data type, a structured data type, or an enumeration.

The UML primitive data types[[4]](#footnote-4) are used: Boolean, Integer, Real, String, and UnlimitedNatural (the latter is only in XMI for the unlimited value of an upper cardinality).

A structured data type can have multiple attributes which are defined by other data types.

Some XML Schema primitive data types[[5]](#footnote-5) are used. They are defined as UML primitive data types and defined semantically by the related XML Schema data type definition. Following XML Schema primitive data types are used: anyURI, date, and language.

All structured data types make finally use of the mentioned four UML primitive data types and three XML Schema primitive data types.

The UML primitive data types can be mapped to XML Schema data types in representations where they exist like in XML Schema and OWL/RDF.

**Mapping of primitive data types[[6]](#footnote-6):**

|  |  |
| --- | --- |
| **UML** | **XML Schema** |
| PrimitiveTypes::Boolean | http://www.w3.org/2001/XMLSchema#boolean |
| PrimitiveTypes::Integer | http://www.w3.org/2001/XMLSchema#integer |
| PrimitiveTypes::Real | http://www.w3.org/2001/XMLSchema#double |
| PrimitiveTypes::String | http://www.w3.org/2001/XMLSchema#string |
| PrimitiveTypes::UnlimitedNatural | http://www.w3.org/2001/XMLSchema#string |

#### Comment

Each item can have a definition which is expressed as UML comment.

### Naming Convention

All items are named according to rules aligned with ISO/IEC 11179-5[[7]](#footnote-7). The names are possible compounds of multiple nouns and adjectives. Instead of a separator, the first letter of each name part within a single name is capitalized (sometimes called CamelCase).

Names of classes, data types, and enumeration literals start with an uppercase letter. Names of associations and attributes start with a lowercase letter.

1. ISO/IEC 11179-6:2015, Information technology - Metadata registries (MDR) - Part 6: Registration, Annex A, Identifiers based on ISO/IEC 6523, <http://standards.iso.org/ittf/PubliclyAvailableStandards/c060342_ISO_IEC_11179-6_2015.zip> [↑](#footnote-ref-1)
2. UML 2.5.1, Semantics of associations, page 200, <https://www.omg.org/spec/UML/2.5.1/PDF> [↑](#footnote-ref-2)
3. The Unified Modeling Language User Guide, Booch, Grady; Rumbaugh, James; Jacobson, Ivar; Reading, Mass., 1999, page 144 [↑](#footnote-ref-3)
4. XMI representation of UML primitives, <https://www.omg.org/spec/UML/20100901/PrimitiveTypes.xmi> [↑](#footnote-ref-4)
5. XML Schema primitive data types: <https://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes> [↑](#footnote-ref-5)
6. UML 2.5.1, XMI Serialization of the PrimitiveTypes model library, page 754, <https://www.omg.org/spec/UML/2.5.1/PDF> [↑](#footnote-ref-6)
7. ISO/IEC 11179-5, Information technology - Metadata registries (MDR) — Part 5: Naming principles, <http://standards.iso.org/ittf/PubliclyAvailableStandards/c060341_ISO_IEC_11179-5_2015.zip> [↑](#footnote-ref-7)