!!!UNDER CONSTRUCTION!!!

International

Virtual

Observatory

Alliance

VO-DML: a proposal for a consistent modelling language for IVOA data models

Version 0.x-20130416

Working Draft 2013 April 16

TODO/TBD

- Update abstract
- Update executive summary.
- Update section 2

This version:
0.x-20130416
Latest version:
0.x-20130412
Previous version(s):

Editors:
Gerard Lemson
Laurent Bourgès

Authors:
UTYPE-s tiger team+Laurent Bourges

Abstract

We propose that data models in the IVOA should be written in a consistent language. In this WD we propose such a language, which we name **VO-DML** (VO Data Modelling Language). VO-DML is a conceptual modelling languae derived from UML and directly borrows concepts from UML's language for describing "Class Diagrams". It was originally conceived as a simplified representation of UMLs XML serialization, XMI. An earlier version of VO-DML has been used extensively in the Simulation Data Model effort[1], and relieved that effort of a lot of work. Extra requirements supported by the current version are the need for a consistent data modelling language to make sense of the UTYPE concept [UTYPES], and the ability to reuse models in a consistent manner.
VO-DML as described in this document is an example of a domain specific modelling language, where the domain is the set of data and meta-data structures handled in the IVOA and Astronomy at large. It has served well in various efforts and provides a rigorous framework form which one can define mappings to various serialization formats [UTYPES]. It provides translational semantics for VOTable annotations using utypes. But it is a custom language, specifically designed for the IVOA data modelling efforts. To investigate whether some existing language might be used instead, in an appendix we provide a short comparison to some existing languages aimed to support domain specific modelling.

Status of This Document

This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress".

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

[1] The language was developed mainly in the VO-URP project (https://code.google.com/p/vo-urp/) and was formally defined in its intermediateModel.xsd document (http://ivoa.net/Documents/SimDM/20120503/uml/intermediateModel.xsd).

Contents

1 Background and motivation

Data models in the IVOA have a particular goal, namely to facilitate interoperability. In particular they should provide a common language to interpret and understand data sets that are published online and allow one to explain the contents of serialized data sets that are used to interchange information.

The form(at) and contents of these data sets are generally not explicitly known; they have generally been created to serve the purposes of the entity/organization that owns them. The owners of the data may be willing to described their data holdings in some standardized form (e.g. TAP), or send their data over the net in some standardized serialization format (e.g. VOTable), but may be unwilling or unable to change the structural design of the data holdings or the contents of the serializations to conform to some model. They may however be able to provide annotations to explain the contents of the data sets they expose.

The IVOA has one standardized way to provide such annotation, namely UCDs. These allow one to annotate data elements with some concept from a simple semantic vocabulary. UTYPEs were supposed to add to this ability by allowing one to identify data structures with elements defined in some data model. A simple interpretation of this was however not available, mainly due to a lack of understanding of the target of these UTYPE pointers.

Here we describe a language for expressing data models that is tailored to provide a target for such annotations and that can provide a consistent, translational semantics to the annotated elements.

This document contains a specification for a data modeling language, or *meta-model*, that all formal data models defined in the IVOA should follow. It consists of a conceptual part and a serialization language. The latter provides an XML format and this specification states that all data models in the IVOA MUST have a representation in that format.
The conceptual part of the spec goes by the name of **VO-DML**, short for VO Data Modeling Language. The serialization language goes by the name of **VO-DML/Schema** and an XML document conforming to this standard defines a data model, and is referred to as the **VO-MDL/XML** representation of that data model.
We also describe a UML Profile [REF] that represents the VO-DML concepts in UML and allows one to provide a graphical representation of each model. This is referred to as **VO-UML**, but is *informative*, *not* a normative part of the spec. These 4 different representations will be discussed and used in the rest of this document.
Here we focus on the motivation for this proposal.
This document finds its origin in the efforts of the UTYPEs tiger team. During the deliberations and analysis of the problem it was charged with, it was concluded that an important ingredient to its resolution was to put the IVOA data modeling on a firmer and more formal basis. In particular it became clear that to interpret utypes, one need to make data models first class citizens in a sense we will now describe.
Most data models in the IVOA had been designed in an ad hoc fashion, with the precise specification of a data model left to each individual effort. Generally the most formal result was a serialization format for instances of the data model. This could be in the form of an XML schema, or a VOTable dialect, or a TAP_SCHEMA defining the storage of instances in a relational database. Even when using a common format such as XSD, thanks to the large redundancy of concepts in that language, there was a large variety of ways by which different models were expressed.
Such heterogeneity is a problem for a specification of utypes, which were supposed to be a "pointer into a data model" that can be used by code to infer information about data serialized in some general manner. Though a syntax for utypes was proposed at some point [REF to early utypes doc], in contrast to the original source of the grammar [REF to SimDM], there was no formal language provided that gave meaning to the components in the grammar.
I.e. there was no formal understanding of *what* precisely a utype used in some meta-data annotation could point at, preventing a possible understanding of what such an annotation might mean.
The approach pioneered in the SimDM effort however *did* offer such an interpretation. It was based on a formal data modeling language. This language, derived from UML, defines some core modeling concepts that can be mapped to all serializations, and provides a common representation to interpret these. It is

implementation neutral and included explicit identifiers to the various data modeling elements. These can serve as the targets of the pointers that utypes were supposed to be.
The so called UTYPEs mapping document [REF] works out this mapping in great detail and provides constraints and semantics to such mappings. The current document provides the definition of the meta-model itself.
One important further benefit of the particular proposal is that it explicitly and naturally allows the possibility of model *reuse*. I.e. the efforts of one modeling effort can be easily reused in that of another, even though the final goal of one might be the creation of an XML serialization format, and the other a relational data model. The actual derivation of these representations can in principle be fully automated as the VO-URP effort has shown. That project was a side result of the SimDM effort and provides a generation pipeline deriving RDB, XSD, JAVA and HTML representations of a model represented in an earlier version of VO-MDL/XML.
VO-URP showed that it is possible from such a specification alone, one that could be hand written without undue problems, to generate a (web) service that allows access to a TAP compatible database automatically created from the model, which can ingest and deliver XML representations of the stored objects that follow the standard XML Schema representation.

2 VO-DML, VO-UML, VO-DML/Schema and VO-DML/XML

In this specification we distinguish between the conceptual meta-model, **VO-DML** and the XML based serialization language for expressing data models, which we refer to as **VO-DML/XML**. The format and contents of a valid VO-DML/XML file is prescribed by an XML schema plus schematron file which we denote together as **VO-DML/Schema**. Their relation between VO-DMl and VO-DML/XML is equivalent to the relation between UML [REF] and representations of a UML data models in the serialization format XMI [REF].
In fact VO-DML is directly derived from UML, more exactly it can be interpreted as being a UML *Profile* [REF], a domain specific "dialect" of UML. We actually use a "real" UML Profile in the UML modeling tool with which we create graphical examples of models. This profile is complete, i.e. is able represent all elements of VO-DML. We will refer to this profile as **VO-UML**[2] and show how to use it in the examples below. We discuss these different components of the specification in the next subsections.

2.1 VO-DML + VO-UML

VO-DML defines the concepts used to create data models in the IVOA. It uses a subset of the components from UML Class Diagrams and hence follows an object-oriented approach, but restricts itself to structure. Operations are explicitly

[2] Omar Laurino suggested this name.

excluded from our language. Constraints are supported, but in a restricted manner.
VO-DML has been expressed as a UML profile, VO-UML, using *stereotypes* with *tag definitions* to enable modeling of domain specific components in the graphical tool.
But note, though we use UML for illustrations, and have used it to define some models, VO-DML is *not dependent* on UML or such tools. VO-DML data models need not have a UML representation, but they MUST have a serialization in terms of VO-DML/XML (see section 5 for details on how to define data models).
Hence the VO-UML part in this spec is INFORMATIVE, not NORMATIVE [TBD is this correct usage of these terms?].

2.2 VO-DML/XML + VO-DML/Schema

For most use cases a VO-DML data model must be serialized in a computer readable format. The serialization language to be used for this is VO-DML/XML. VO-DML/XML is XML following a formal syntax defined by an XML schema vo-dml.xsd [3] and further constrained by an associated Schematron file, vo-dml.sch.xml[4]. These files implement all the concepts described in section 3 . The schema files are self-documented as much as possible, but here we give a few details of the overall design, focusing on technical aspects of the implementation.

VO-DML/XML is a simple representation of a VO-DML model as an XML document. This introduction of a custom designed serialization language rather than using some existing language could be seen as an unnecessary complication. We *could* for example also consider using XMI as the standard for serializing VO-DML models. However, XMI is a rather unwieldy format that hides many of the features we want to make explicit. Hence as a language from which to derive information of the model without very sophisticated tools it is ill suited. Also we do not assume all users have access to a UML modeling tool that can support all the UML modeling features we need to create VO data models, and hand editing XMI is nigh impossible. We also foresee that users may want to derive models from other representations (e.g. XML schema, RDF; see SimTAP discussions) and XMI as target language for such a tool requires deep understanding of its format.
We do think XML is a useful language for serializations, in particular because, being completely under control of the IVOA DM WG, it can be tailored to the requirements deriving from its usage in the IVOA. We have more freedom to restrict the format and implement the appropriate constraints. In VO-DML/XML the format and constraints are explicitly and formally defined using XML schema and Schematron [REF]. Hence VO-DML/XML files must conform to an XML schema file (<root-url>/vo-dml.xsd) that defines the *structure* of valid serializations, and a Schematron file (<root-url>/vo-dml.sch.xml) that defines additional constraints on its contents. We refer to this implementation of the

[3] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.xsd
[4] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.sch.xml

modeling language as VO-DML/Schema. VO-DML/Schema is a direct implementation of the UML profile; it exposes all modeling concepts explicitly, and ignores the many UML/XMI features that are not needed.
VO-DML/Schema is compared to alternative languages and similar approaches to designing domain specific modeling languages in Appendix B.

2.3 Other representations

See section 4 for a discussion on how one might represent data models in application specific formats. These can be considered physical model that should implement the logical model defined in VO-DML. One special case there concerns a possible addition to our family, namely something we might call VO-DML/I, and instantiation format tailored for VO-DML models. This is discussed in section 4.6 .

3 Modeling Concepts and Serialization Language

In the following sub-sections we discuss the different components of the meta-model. We describe their meaning, we indicate where and how they are implemented in the VO-DML/Schema and how they are represented in the VO-UML profile, and we give examples of a VO-DML/XML serialization and a graphical representation. The examples are extracted from a sample data model that is designed to illustrate most of the modeling components. It models astronomical sources and is explained in Appendix A. Its VO-DML/XML representation can be found in the Volute project in GoogleCode[5] [TBD other location?]. The VO-DML/Schema snippets only indicate the start of a XML schema definition, the full details can be found in the corresponding schema documents.
Most modeling concepts have a 1-1 relation between the VO-DML, VO-DML/Schema and VO-UML representation. In the few cases where we have to treat one of these specially we will indicate this explicitly.
When referring to a VO-DML *concept* we will use **boldface**. When referring to an XML schema implementation of a concept we will use `courier` font. When referring to a UML concept we will use *italics*. When using UML examples, screenshots from a UML modeling tool or snippets of XMI, we have used MagicDraw[6] Community Edition 12.1, which supports UML 2.0[7], serializing its models to XMI 2.1[8].

3.1 Model

A (data) model represents a coherent set of type definitions, by which it represents the concepts that have an explicit place in its universe of discourse,

[5] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/source/SourceDM.vo-dml.xml
[6] http://www.nomagic.com/products/magicdraw.html, edition CE 12.1 is no longer available online.
[7] http://schema.omg.org/spec/UML/2.0
[8] http://schema.omg.org/spec/XMI/2.1

i.e. which concepts one can talk/"discourse" about. Each data model is generally represented by a single document. **Model** is an explicit concept in VO-DML and its representations, for we need to be able to refer to models explicitly in the language.

VO-UML

In a UML diagram the **Model** is represented by a complete XMI document, with the element representing it showing up in the root of the model. The VO-UML profile contains a <<model>> *stereotype* which defines *tags* which allow one to define extra metadata about the model and which correspond to the metadata elements defined in the subsections below.

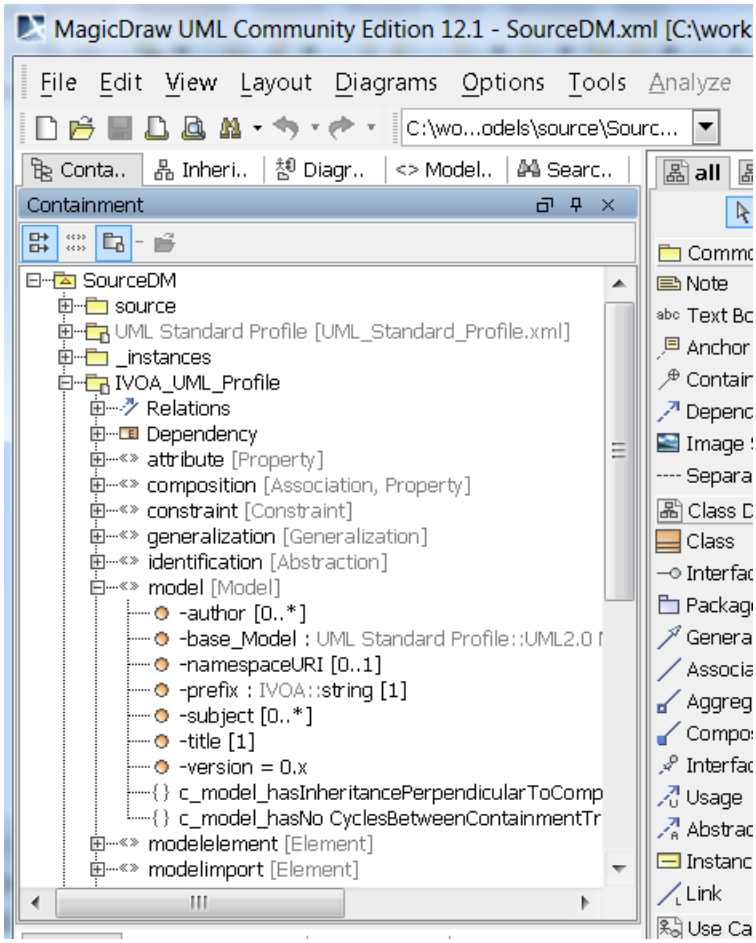
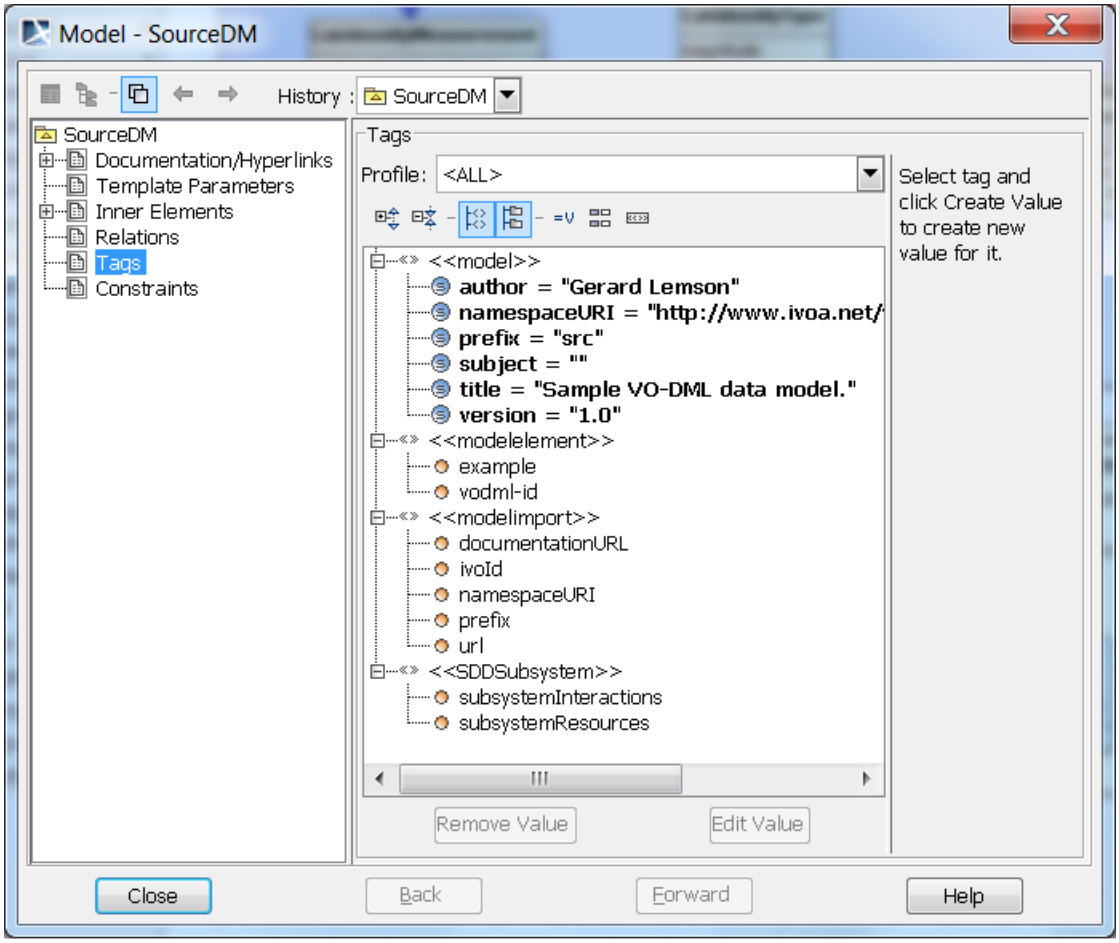


Figure 1 Root of a UML document defining the example model, SourceDM. Note the use of the IVOA_UML_PROFILE with its various *stereotype* definitions. In particular <<model>> defines various *tags* corresponding to the meta data elements for a Model**.**

The tags can be given values as shown in the following example:



VO-DML/Schema

In VO-DML/Schema **Model** is represented by a complexType `Model`, containing definitions for meta-data elements and collections of type definitions, possibly distributed over packages. It defines the type for the only possible root element in a VO-DML/XML document, named `model`.

```
<xsd:complexType name="Model">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="1"/>
    <xsd:element name="description" type="xsd:string"
         minOccurs="0"/>
...
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="model" type="Model">
  <xsd:unique name="unique_ids">
    <xsd:selector xpath=".//vodml-id" />
    <xsd:field xpath="." />
  </xsd:unique>
</xsd:element>
```

VO-DML/XML

```
<vo-dml:model
      xmlns:vo-dml="http://volute.googlecode.com/dm/vo-dml/v0.9">
 <name>SourceDM</name>
 <description>This is a sample data model. ...
 </description>
 <title>Sample VO-DML data model.</title>
 <prefix>src</prefix>
 <namespaceURI>
   http://www.ivoa.net/vo-dml/models/SourceDM#
 </namespaceURI>
 <version>0.x</version>
 <lastModified>2013-05-04T19:24:52</lastModified>
...
```

Model has the following components, which have a 1-1 correspondence in the VO-DML/Schema. See there for more extensive comments.

3.1.1 name : string [1]
The (short) name of the model.

3.1.2 description : string [0..1]
A human readable description of the model.

3.1.3 title : string [0..1]
Formal, long, title of this data model.

3.1.4 namespaceURI: anyURI[0..1]
TBD

3.1.5 prefix: ModelPrefix [1]
TBD

3.1.6 author: string[0..*]
List of names of authors who have contributed to this model.
TBD could be expanded to a Curation like model.

3.1.7 version : string [1]
Label indicating the version of this model.

3.1.8 previousVersion : anyURI [0..1]
This attribute contains a URL[TBD URI?] identifying a VO-DML/XML document representing the previous version of this data model, from which the current version was derived.

3.1.9 import : ModelProxy[→] [0..*]
For a **Model** to reuse types from another model, or to use other of its elements, that model must be "imported" explicitly. An imported model is represented by a

ModelProxy that specifies some of the metadata of the model as well as how it is to be used.

VO-DML/XML

TBD check/fix

```
<vo-dml:model>
...
  <import>
    <name>PhotDM-alt</name>
    <url>https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/photdm-alt/PhotDM-alt.vo-dml.xml</url>
    <prefix>photdm-alt</prefix>
    <documentationURL>https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/photdm/PhotDM.html</documentationURL>
  </import>
...
```

3.1.10 package : Package[→] [0..*]

A Model can distribute its type definitions over packages. This provides for name spacing options, allowing multiple types with the same name.

VO-DML/XML

```
<vo-dml:model>
...
  <package>
    <vodml-id>source/</vodml-id>
    <name>source</name>
...
```

3.1.11 objectType : ObjectType[→] [0..*]

Collection of ObjectType-s defined directly under the model. In many IVOA data models packages have not been explicitly defined. Instead types were defined directly under the model. In this meta-model we support this as well by adding collections for each of the different “types of types”.

3.1.12 dataType : DataType[→] [0..*]

Collection of DataType-s defined directly under the model.

3.1.13 primitiveType : PrimitiveType[→] [0..*]

Collection of PrimitiveType-s defined directly under the model.

3.1.14 enumeration : Enumeration[→] [0..*]

Collection of Enumeration-s defined directly under the model.

3.1.15 ModelProxy

A Model can import another model. This implies generally that elements of the external model are used in the definition of elements in the current model. The external model must be represented by a ModelProxy. This "proxy" provides metadata allowing one to access the remote model and its documentation, as well as a prefix that must be used when referring to elements in the remote model.

VO-UML

A ModelProxy is represented by a child *Model* element with special stereotype <<modelimport>>. Graphically and possibly some contained type proxies. The latter MUST define a value for the 'vodml-id' tag.

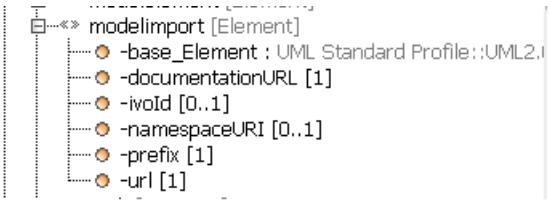


Figure 2 Stereotype definition for <<modelimport>>.

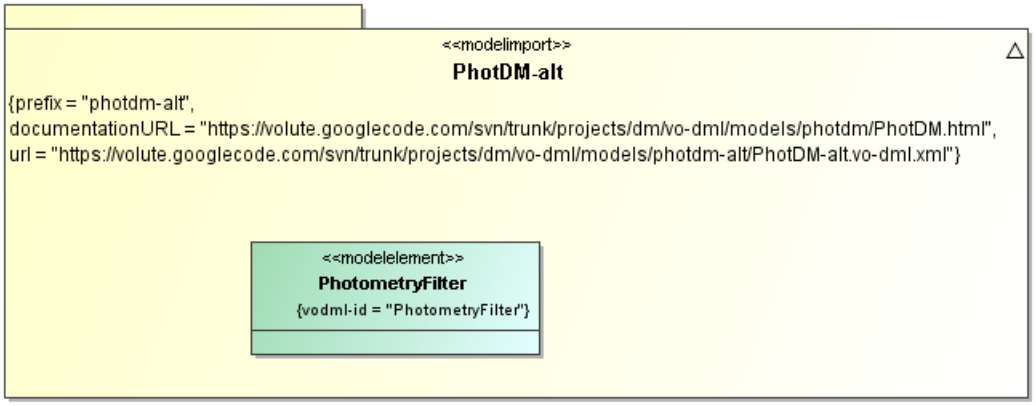


Figure 3 Graphical representation of an imported model. Note the usage of the stereotype <<modelimport>> and the values assigned to the various tags.

VO-DML/Schema

```
<xsd:complexType name="ModelProxy">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" minOccurs="0"/>
      <xsd:element name="ivoId" type="xsd:anyURI" minOccurs="0"/>
      <xsd:element name="url" type="xsd:anyURI" />
      <xsd:element name="prefix" type="ModelPrefix" />
      <xsd:element name="documentationURL" type="xsd:anyURI" />
    </xsd:sequence>
  </xsd:complexType>
```

VO-DML/XML

```
  <import>
    <name>PhotDM-alt</name>
    <url>
     https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/photdm-alt/PhotDM-alt.vo-dml.xml
    </url>
    <prefix>photdm-alt</prefix>
    <documentationURL>
https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/photdm/PhotDM.html
    </documentationURL>
  </import>
```

3.2 Referencing elements

A data model consists of model elements of various types. In the VO-DML/XML serialization almost all of these must have an explicit identifier element that makes it possible for them to be explicitly referenced, either from inside the model, or from an external context. VO-DML/Schema defines a special types that represents these referencable elements, and a type that indicates how to refer to such elements inside VO-DML.

VO-DML/Schema

All elements (apart from `Model`) discussed in this section extend `ReferencableElement`. This abstract base class has an identifier element `<vodml-id>`, the value of which must be unique in the model. This means that these elements can be referenced using this identifier. In VO-DML this is used explicitly in the `ElementRef` type that represents such references inside the meta-model using a `<utype>` element. The type definitions for vodml-id and utype are restricted strings, the details of which are TBD.

```
<xsd:simpleType name="ElementID" >
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[\w\./_*]+"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="ReferencableElement" abstract="true">
  <xsd:sequence>
    <xsd:element name="vodml-id" type="ElementID " minOccurs="1"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:simpleType name="ModelPrefix">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[\w_-]+"/>
  </xsd:restriction>
```

```
</xsd:simpleType>

<xsd:simpleType name="UTYPE">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[\w_-]+:[\w\./_*]+" />
   </xsd:restriction>
</xsd:simpleType>
```

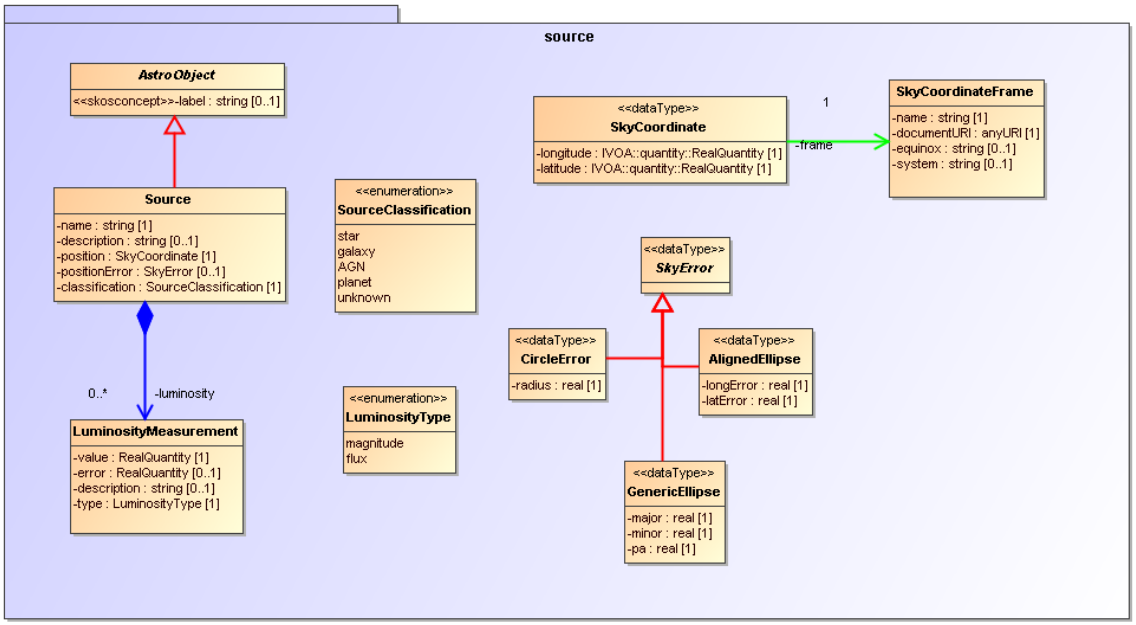
VO-DML/XML

3.3 Package

Packages divide the set of types in a model in subsets, providing these with a common namespace. Their names must be unique in this context only. A package may contain child packages.
This concept is equivalent to the UML *Package* and similar to an XML namespace or a Java package.

VO-UML

TBC A Package is represented by the UML *Package* element



VO-DML/Schema

In VO-DML/Schema, **Package** is represented by a complex type of the same name and extends `ReferencableElement`.

```
<xsd:complexType name="Package">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <xsd:sequence>
        <xsd:element name="objectType" type="ObjectType" minOccurs="0"
```

```
            maxOccurs="unbounded"/>
        <xsd:element name="dataType" type="DataType" minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="enumeration" type="Enumeration"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="primitiveType" type="PrimitiveType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="package" type="Package" minOccurs="0"
            maxOccurs="unbounded"/>
...
</xsd:complexType>
```

VO-DML/XML

```
<package>
  <vodml-id>source</vodml-id>
  <name>source</name>
  <description>...</description>
  <objectType>
    <vodml-id>source.LuminosityMeasurement</vodml-id>
    <name> LuminosityMeasurement</name>
...
```

A package has separate collections for each of the type classes. This avoids the need for xsi:type casting in serializations and facilitates tracing path expressions.

3.3.1 objectType : ObjectType[→] [0..*]
Collection of ObjectTypes defined in this package.

3.3.2 dataType : DataType[→] [0..*]
Collection of DataTypes defined in this package.

3.3.3 primitiveType : PrimitiveType[→] [0..*]
Collection of PrimitiveTypes defined in this package.

3.3.4 enumeration : Enumeration[→] [0..*]
Collection of Enumerations defined in this package.

3.3.5 package : Package[→] [0..*]
Collection of child packages defined in this package.

3.4 Type
The goal of a VO-DML data model is to define **Types**. A type expresses a particular concept in a formal, machine usable manner. Its definition in a data model expresses that that concept is important in the universe of discourse covered/defined by the data model. It makes the concept part of the formal vocabulary defined by the model. It classifies "instances"/"objects"/"values" in the world into groups defined by common properties and meaning. Every instance

"worth talking about" "has a"/"is declared to have a" type. Every instance has exactly one type, though due to inheritance an instance is also an instance of any base type of the declared type.

VO-DML/Schema

An abstract complexType named `Type` is introduced that is the base class of all more concrete type definitions. It extends `ReferencableElement`, hence all type definitions can be referenced and MUST have a `<vodml-id>` element. Types may be abstract, in which case no instances can be produced (similar to for example abstract classes in Java. They may also `extend` another type, which will be referred to as the base type. The base type is identified by a `utype`.

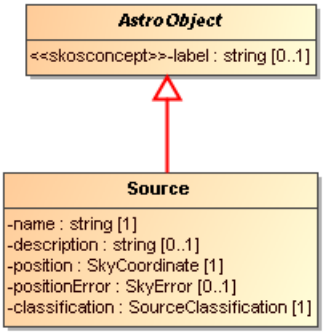
```
<xsd:complexType name="Type" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <xsd:sequence>
        <xsd:element name="extends" type="ElementRef" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="abstract" type="xsd:boolean"
         default="false" use="optional" />
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

3.4.1 Inheritance

Indicates the typical “is a” relation between the sub-type and its base-type (the one pointed at). In VO-DML we do not support multiple inheritances. Furthermore ObjectTypes extend ObjectTypes, DataTypes extend DataTypes etc.

VO-UML

Inheritance relationship is represented by a UML generalization arrow from subclass to base class. The arrow is



VO-DML/XML

```
    <objectType>
      <vodml-id>source/Source</vodml-id>
      <name>Source</name>
```

```
...
      <extends>
        <utype>src:source/AstroObject</utype>
      </extends>
...
```

3.5 Value Type

The most important categorization of **Types** is that between so called **object types** and **value types**. [TBC find equivalent categorizations: reference type vs value type, ...] A **ValueType** represents a simple concept that is used to describe/define more complex concepts such as ObjectTypes. In contrast to ObjectTypes, instances of ValueType-s, i.e. *values*, need not be explicitly identified. They are identified by their value. For example an integer is a value type; all instances of the integer value '3' represent the same integer.
The domain of a value type, i.e. its set of valid instance/values, is *self-evident from its definition*. Hence also the existence of particular values is self-evident and therefore needs not to be explicitly stated. Also this is in contrast to the case of ObjectTypes discussed below.

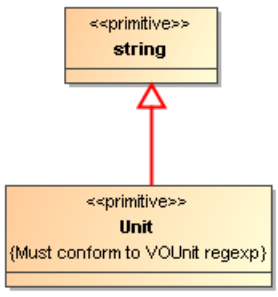
VO-DML/Schema

```
<xsd:complexType name="ValueType" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="Type">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

3.6 PrimitiveType

A PrimitiveType represents an atomic piece of data, a *value*. Examples are the standard types like integer, Boolean, real, and string (which we treat as an atomic value, *not* an array of characters), integer and so on.
A primitive type can be an extension of another primitive type, but must then always be considered a restriction on the possible values of that type.
.

VO-UML



VO-DML/XML

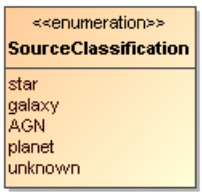
```
<primitiveType>
```

```
  <vodml-id>quantity/Unit</vodml-id>
  <name>Unit</name>
  <description>
    Must conform to definition of unit in VOUnit spec.
  </description>
  <extends>
    <utype>ivoa:string</utype>
  </extends>
</primitiveType>
```

3.7 Enumeration

An Enumeration is a PrimitiveType with a finite list of possible values, the Literals. This list restricts the domain of possible values which are to be treated as strings.

VO-UML



VO-DML/XML

```
    <enumeration>
      <vodml-id>source/SourceClassification</vodml-id>
      <name>SourceClassification</name>
      <literal>
        <vodml-id>source/SourceClassification.star</vodml-id>
        <name>star</name>
        <description>...</description>
        <value>star</value>
      </literal>
      <literal>
        <vodml-id>source/SourceClassification.galaxy</vodml-id>
        <name>galaxy</name>
        <description>...</description>
        <value>galaxy</value>
      </literal>
...
```

3.7.1 Literal

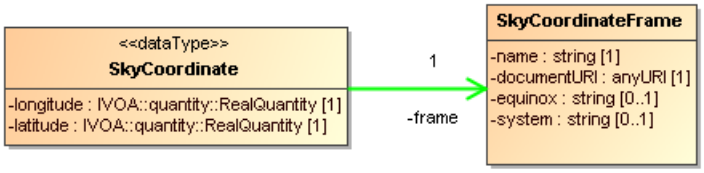
3.8 DataType

A DataType is a value type with structure. The structure is generally defined by attributes on the DataType, and possibly references. The state of instance of a DataType, i.e. a *value*, consists of the assignment of values to all the attributes and references. This is similar to ObjectTypes defined below, but in contrast to ObjectTypes, datatypes have no explicit identity. Also, DataType-s are defined by their state only. I.e. two DataType instances with the same state are the same

instance. Instead, ObjectTypes with the same state but different identity are not the same
For example the DataType Position3D, with attributes x, y and z, is completely defined by the values of the three attributes. There are no 2 distinct instances of this DataType with exact same values (x=1.2,y=2.3,z=3.4).
Note, we are adding the ability to add outgoing references to DataType. This makes certain patterns more reusable. The reference is assumed to provide reference data wrt which the rest of the value should be interpreted. For example a SkyCoordinate may have a reference to a SkyCoordinateFrame to help interpret the values of the longitude/latitude attributes. [TBD check proper STC example? Or use other example?].

VO-UML

This concept is directly derived for UML's Data Type[9]. It is represented by a box with stereotype <<datatype>> and possibly attributes.



VO-UML/Schema

```
<xsd:complexType name="DataType">
  <xsd:complexContent>
    <xsd:extension base="ValueType">
      <xsd:sequence>
        <xsd:element name="attribute" type="Attribute"
             minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="reference" type="Reference"
             minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

VO-UML/XML

```
<dataType>
  <vodml-id>source.SkyCoordinate</vodml-id>
  <name>SkyCoordinate</name>
  <description>...</description>
  <attribute>
    <vodml-id>source.SkyCoordinate.longitude</vodml-id>
    <name>longitude</name>
    <description>...</description>
```

9 http://pic.dhe.ibm.com/infocenter/rsarthlp/v8/index.jsp?topic=%2Fcom.ibm.xtools.modeler.doc%2Ftopics%2Fcdatatypes.html

```
    <datatype>
      <utype>ivoa:quantity.RealQuantity</utype>
    </datatype>
    <multiplicity>1</multiplicity>
  </attribute>
  <attribute>
    <vodml-id>source.SkyCoordinate.latitude</vodml-id>
    <name>latitude</name>
    <description>...</description>
    <datatype>
      <utype>ivoa:quantity.RealQuantity</utype>
    </datatype>
    <multiplicity>1</multiplicity>
  </attribute>
  <reference>
    <vodml-id>source.SkyCoordinate.frame</vodml-id>
    <name>frame</name>
    <description>...</description>
    <datatype>
      <utype>src:source.SkyCoordinateFrame</utype>
    </datatype>
    <multiplicity>1</multiplicity>
  </reference>
</dataType>
```

3.8.1 attribute: Attribute[→] [0..*]

Collection of Attribute definitions.

3.8.2 reference: Reference[→] [0..*]

Collection of Reference definitions. A reference on a DataType is assumed to provide reference data to help interpreting.

3.9 OjectType extends Type

ObjectTypes are the fundamental building blocks of a data model. An ObjectType represents a full-fledged concept and is built up from properties and relations to other ObjectTypes. An important feature of ObjectTypes as opposed to ValueTypes (see below) is that instances of ObjectTypes, i.e. objects, have their own, explicit identity[10]. That is, we want to assign an explicit identifier to each particular usage of this concept, for instance here to distinguish between various Experiment instances.

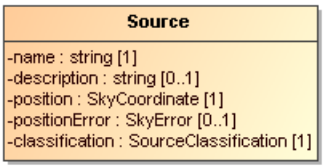
Another feature of ObjectType-s is that the existence of certain instances is *not* self-evident from their definition. For example the definition of the ObjectType Person does not mandate that a certain person with name="alice" and age=23 years exists. Hence it is meaningful to list instances of ObjectType-s explicitly to

[10] This is admittedly a somewhat theoretical but important object-oriented concept.

"announce" their existence. This is indeed why we have serializations of data models in the first place, to announce the existence of objects of various types.

VO-UML

An ObjectType is represented in VO-UML by a UML *Class*, a box with a name and possibly a *stereotype* such as <<modelelement>>. An ObjectType may have attributes and be the source or target of relationships..



VO-DML/Schema

In XSD the **ObjectType** is represented by a complexType definition `ObjectType` that extends `Type`.

```
<xsd:complexType name="ObjectType">
  <xsd:complexContent>
    <xsd:extension base="Type">
    <xsd:sequence>
        <xsd:element name="container" type="Container"
             minOccurs="0" maxOccurs="1"/>
        <xsd:element name="attribute" type="Attribute"
             minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="collection" type="Collection"
             minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="reference" type="Reference"
             minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

VO-DML/XML

```
<objectType>
  <vodml-id>source.Source</vodml-id>
  <name>Source</name>
  <description>...</description>
  <extends>
    <utype>src:source.AstroObject</utype>
  </extends>
  <attribute>
    <vodml-id>source.Source.name</vodml-id>
    <name>name</name>
    <description>...</description>
    <datatype>
      <utype>ivoa:string</utype>
    </datatype>
    <multiplicity>1</multiplicity>
  </attribute>
```

```
  <attribute>
...
```

3.9.1 container: Container[→] [0..1]

Pointer to the ObjectType that is the parent in a collection of which this ObjectType is the declared child datatype.
[TBD really redundant, mainly here so a foreign key has something to identify with. Remove and use vo-dml:ObjectType.CONTAINER instead?]

3.9.2 attribute: Attribute[→] [0..*]

Collection of Attribute definitions.

3.9.3 reference: Reference[→] [0..*]

Collection of Reference definitions.

3.9.4 collection: Collection[→] [0..*]

Collection of collection definitions.

3.10 Role extends ReferencableElement

A **Role** represents the usage of one type (call it "target") in the definition of another (type "source"). The "target" type is said to play a role in the definition of the "source" type. Examples are where the target is the base class of the source, or where the target is the data type of an attribute defined on the source.
There are different kinds of roles, in VO-DML defined as sub types of *Role*. *Role* defines only a "datatype" attribute that has an ElementRef as data type, but is constrained by Schematron rules to reference a Type. Specializations of Role will introduce further constraints.

```
<xsd:complexType name="Role" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <xsd:sequence>
        <xsd:element name="datatype" type="ElementRef"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

VO-DML/Schema

Role is only explicitly represented in the VO-DML/Schema. It defines the `datatype` reference that identifies (through a `<utype>`) the type that is playing the role on the parent type containing the role. `Role` is abstract, hence only subclasses can be instantiated.

```
<xsd:complexType name="Role" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <xsd:sequence>
```

```
        <xsd:element name="datatype" type="ElementRef"/>
        <xsd:element name="multiplicity" type="Multiplicity"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

3.10.1 datatype : ElementRef

The **datatype** property of a Role identifies the target type of the role, the one which actually "plays the role". In VO-DML/Schema it is represented by an `ElementRef` that MUST identify a `Type`.

3.10.2 multiplicity : Multiplicity

Indicates the multiplicity or cardinality of the role. This indicates how many instances of the target **datatype** can be assigned to the role property.

3.11 Attribute extends Role

An **Attribute** is the role a value type can play in the definition of a structured type, i.e. an **ObjectType** or **DataType**. It represents a typical property of the parent type such as age, mass, length, position etc.
Rules:

- The datatype of an Attribute, inherited from Role, MUST be a **ValueType**.

VO-UML

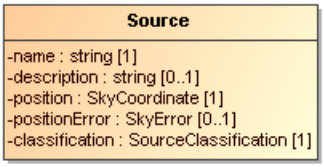


Figure 4 The rows in the lower part of the box represent attributes. Their name, datatype and multiplicity are indicated.

VO-DML/Schema

```
<xsd:complexType name="Attribute">
  <xsd:complexContent>
    <xsd:extension base="Role">
      <xsd:sequence>
        <xsd:element name="constraints" type="Constraints"
             minOccurs="0"/>
        <xsd:element name="skosconcept" type="SKOSConcept"
             minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
```

VO-DML/XML

3.11.1 constraints : Constraints [0..1]

3.11.2 skosconcept : SKOSConcept [0..1]

If an Attribute defines a "skosconcept", it indicates that its values should represent a SKOS concept [TBD add reference]. It implies the value of the attribute in an instance should be a URI [TBD maybe just a preferredValue ...?] identifying a concept in some SKOS vocabulary that fulfills the constraints of the SKOSConcept definition. In this case the data type attribute should be compatible with a string.

3.12 Constraints

TBD

3.13 Relation extends *Role*

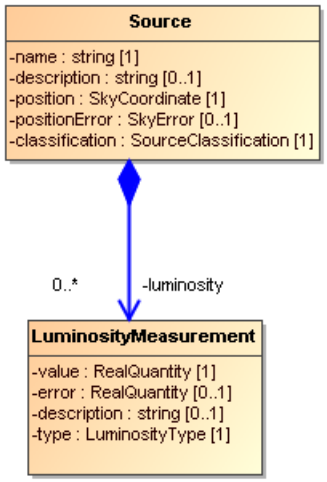
A **Relation** is a role played by an **ObjectType** in the definition of either another ObjectType or a DataType. It indicates the ObjectType that is the target (datatype) of the relation is related in some fashion to the source type.

3.14 Collection extends Relation

An ObjectType may be "composed of" other object types. A **Collection** represents this composition relationship between a parent and child ObjectType. The life cycles of the child objects are governed by that of the parent.

VO-UML

In VO-UML a collection relation is represented by a composition association, an arrow with a closed diamond indicating a composition side of the container and an arrow on the end of the contained class. We generally use a blue colour for this relation.



VO-DML/XML

3.15 Reference extends Role

A reference is a relation that indicates a kind of *usage*, or *dependency* of one object (the *source*, or *referrer*) on another (the *target*). Such a relation may in general shared, i.e. many referrer objects may reference a single target object.

In general a reference relates two ObjectTypes, but a DataType-s can have a reference as well. An example of this is a coordinate on the sky consisting of a longitude and latitude, which requires a reference to a CoordinateFrame for its interpretation. I.e. the frame is used as "reference data".

VO-UML

A reference is indicated by a (green) arrow from referrer (an ObjectTYpe or DatType) to the target (an ObjectType). In UML an association is used, though the reference is actually most similar to a binary association *end*.

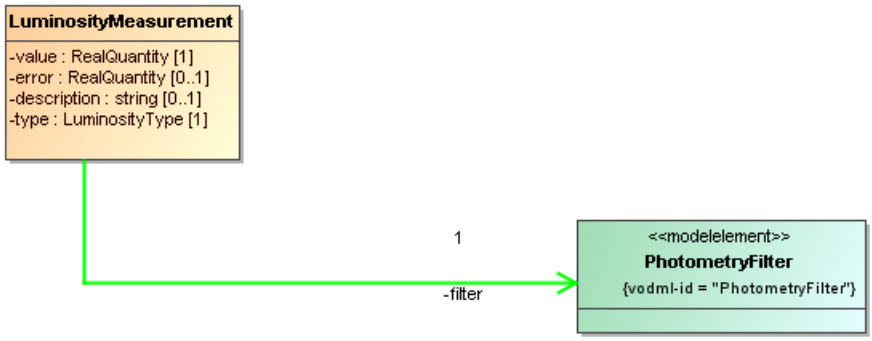


Figure 5 Reference (green arrow) from an ObjectTYpe to an ObjectType.

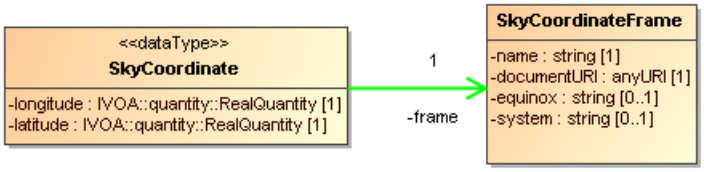


Figure 6 Reference from a DataType to an ObjectType

VO-DML/XML

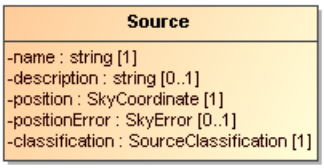
```
    <dataType>
      <vodml-id>source/SkyCoordinate</vodml-id>
      <name>SkyCoordinate</name>
...
      <reference>
        <vodml-id>source/SkyCoordinate.frame</vodml-id>
        <name>frame</name>
        <description>
...
        </description>
        <datatype>
          <utype>src:source/SkyCoordinateFrame</utype>
        </datatype>
        <multiplicity>1</multiplicity>
      </reference>
...
```

3.16 Multiplicity

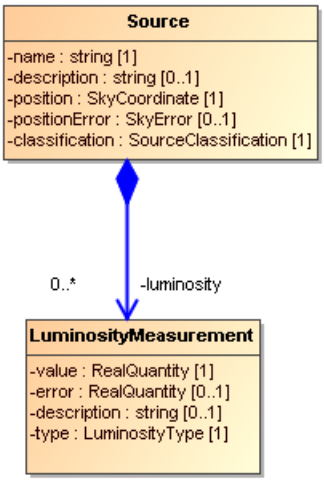
Multiplicity indicates the cardinality of roles.We model this similar to the way this is done in XML schema, namely with a minOccurs/maxOccurs pair of values. The formar indicates the minimum number of instances or values that can be assigned to a given role, the latter the maximum number. Also XMI supports two values and we follow it in using -1 (or any negative value) as a possible value for maxOccurs that indicates that there is no limit on the possible number of instances. In XML schema this is indicated using the string value 'unbounded', in UML diagrams generally with a '*'.
[TBD We may wish to restrict the possible values for maxOccurs for Attribute declarations to *not* allow negative values. I.e. the size of possible value collections represented by an attribute with maxOccurs > 0 should always be fixed. Motivation length, later...]

VO-UML

In VO-UML the cardinality, when assigned to an attribute, shows up in square brackets after the attribute's type. If minOccurs and maxOccurs have the same value, that single value is shown. If they have different values they show up separated by '..'. The value of -1 for maxOccurs is represented by a '*':



When the multiplicity is assigned o a relation, a similar pattern is shown near the name of the relation, close to the target datatype of the relation:



Note that the 0..* here indicates minOccurs=0, maxOccurs=-1.

VO-DML/Schema

```
  <xsd:complexType name="Multiplicity">
    <xsd:sequence>
      <xsd:element name="minOccurs" type="xsd:nonNegativeInteger"
           default="1"/>
      <xsd:element name="maxOccurs" type="xsd:int" default="1"/>
    </xsd:sequence>
  </xsd:complexType>
```

VO-DML/XML

```
<objectType>
  <vodml-id>source.Source</vodml-id>
  <name>Source</name>
...
  <attribute>
    <vodml-id>source.Source.name</vodml-id>
    <name>name</name>
...
    <multiplicity>
      <minOccurs>1</minOccurs>
      <maxOccurs>1</maxOccurs>
    </multiplicity>
  </attribute>
...
   <collection>
     <vodml-id>source.Source.luminosity</vodml-id>
     <name>luminosity</name>
```

```
...
     <multiplicity>
       <minOccurs>0</minOccurs>
       <maxOccurs>-1</maxOccurs>
     </multiplicity>
   </collection>
 </objectType>
```

4 Mapping to other serialization formats

[Here we can discuss how one might map a VO-DML data model to a physical representation format. This mapping would define how instances of the data model can be serialized in the target representation.

4.1 XSD

ObjectType	complexType
DataType	complexType
Enumeration	simpleType with restriction list elements for the EnumL:iterals
PrimitiveType	simpleType, possibly with restriction
Attribute	Element on complexType, type corresponding to mapping of datatype
Collection	Element on complexType, , type corresponding to mapping of datatype
Reference	Element on complexTypwe, type must be able to perform remote referencing
Extend	xsd:extension of type definition

4.2 RDB

[Follow typical Object-Relational mapping rules.

ObjectType	Table
DataType	Implicit, one or more columns in a table
Enumeration	Column in a table
PrimitiveType	simpleType, possibly with restriction
Attribute	One or more columns in a table depending on type
Collection	Foreign key
Reference	Element on complexTypwe, type must be able to perform remote referencing

4.3 Java

[TBD]

4.4 VOTable

[This is the purpose of the UTPE mapping document in …]

4.5 RDF schema

…

4.6 VO-DML/I: A default serialization language?

For examples it may be useful to define an explicit default serialization language that is explicitly tailored to the VO-DML meta-model, rather than to a particular implementation context. It should be able to express explicitly a set of instances of a data model, and od so in terms of the meta-model itself. Its use could be to make the mapping discussion more explicit, for in many cases there are osme decisions that must be made that rely on elements that are not explicitly part of a data model, but are implied by its definition in terms of VO-DML.
An example of this is the fact that it is assumed that every *object*, i.e. instance ObjectType, has an *identifier* that identifies it. This element is never mentioned explicitly in a VO-DML data model, and indeed its precise nature often depends on a particular serialization format. But when describing mappings it is often important to be able to define how this element is mapped as well.
A default and explicit serialization language will allow us to investigate the mapping procedure more carefullt,as well as provide implementation neutral example instance documents that can be compared to their expression in a target serialization format.
UML allows something similar to this; it allows one to create Object-s that are explicitly linked to types defined in a data model, with slots for setting values to attributes and links to represent instances of relations.

5 Using VO-DML: How to define a data model in the IVOA

Here we define the procedure for creating an IVOA data model and related resources according to the VO-DML philosophy.

- Decide "Why"?
  Default answer: to allow existing and future databases to describe their contents (at least partially) in a common model. Sometimes, support for a particular application area (protocol). Support faithful serialization of data models in targeted XML documents and annotated serialization in VOTable.

- Decide on universe of discourse: what must be described? How rich should model be?
- Create conceptual/logical model. In drawings on whiteboard ("VO-UML"), then transcribe to VO-DML and/or VO-DML/XML. Define concepts completely, realizing that applications may pick and choose and transform.
- Sometimes, in application contexts: derive one or more physical representations. Use as much as possible standard derivation of VO-DML to target representation.

5.1 Rules ...

A new data model MUST have a VO-DML/XML representation. This is an XML document that MUST be valid wrt. the vo-dml.xsd schema [11] and the rules embodied by the vo-dml.sch.xml [12] Schematron file. How this representation is produced is not important. It may be written by hand, derived from a UML/XMI representation using an XSLT script as in VO--DML or by some other means. The VO-DML/XML document MUST be available online and an accepted version MUST be available in an IVOA registry with a standard IVO Identifier.
An IVOA data model MUST have an HTML document in which each data model element is described and is referencable through a URL consisting of a root URL linking to the HTML document itself followed by a '#" sign and the utype of the documented element. The XSLT script vo-dml2html.xsl MAY (SHOULD?) be used to produce such a document from the VO-DML/XML representation. The HTML document MUST be made available online and the HTML for the accepted version MUST be registerd in an IVOA registry. TBC.
A VO-DML data model can *import* other data models. This allows it to use elements from the other data model in the definition of its own. One particular model is important and SHOULD be imported by all IVOA data modesl, the IVOA_Profile [13]. This model contains a set of predefined data types, mainly primitive types, such as string, boolean, integer and real.
A data model MAY produce an XML schema matching the data model. This schema SHOULD allow one to define XML documents representing instances of the data model in 1-1 mapping. The XML schema MAY (SHOULD?) be derived from the VO-DML representation using a standard mapping as implemented by the XSLT script vo-dml2xsd.xsl [TBD create this schema], or it MAY be written by hand. In either case the XSD elements MUST (where appropriate) contain an

[11] The schema is currently available under http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.xsd .
[12] The schematron file is currently available under http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.sch.xml. A full validation using both schema and schematron file is part of the ant build script under https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/build.xml.
[13] Currently this data model is available under https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/profile/IVOA_Profile.vo-dml.xml , its HTML representation from https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/profile/IVOA_Profile.html .

app-info element identifying the model element that is represented using its utype (format of app-info is TBD).
A data model MAY[14] produce a TAP schema matching the data model. This can be one derived from the VO-DML/XML representation using a standard mapping as implemented by the XSLT script vo-dml2tap.xsl [TODO create this], or may be written by hand. In either case the TAP schema elements MUST (where appropriate) contain a utype identifying the model element that is represented. [TBD for a complete representation a construct similar to the VOTable GROUP may have to be added to the TAP meta-data elements.]

6 References

[UTYPES] utypes mapping document
[UML-Infr2.0] http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/
[UML-Sup2.0] http://www.omg.org/spec/UML/2.0/Superstructure/PDF/
[XMI2.1] http://www.omg.org/spec/XMI/2.1/

Appendix A Example Source data model

We use a simple data model with hopefully some familiarity to the readers as illustration to the various examples. It is a VO-DML representation of the TAP data model. It can be more fully examined in https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/source .
The following figure shows a UML version of the model following the graphical rules explained in section 3 .

[14] Not all data models lend themselves to an Object-Relational Mapping (ORM). Particular for models that are heavy on value types rather than object types such a representation may not be natural.

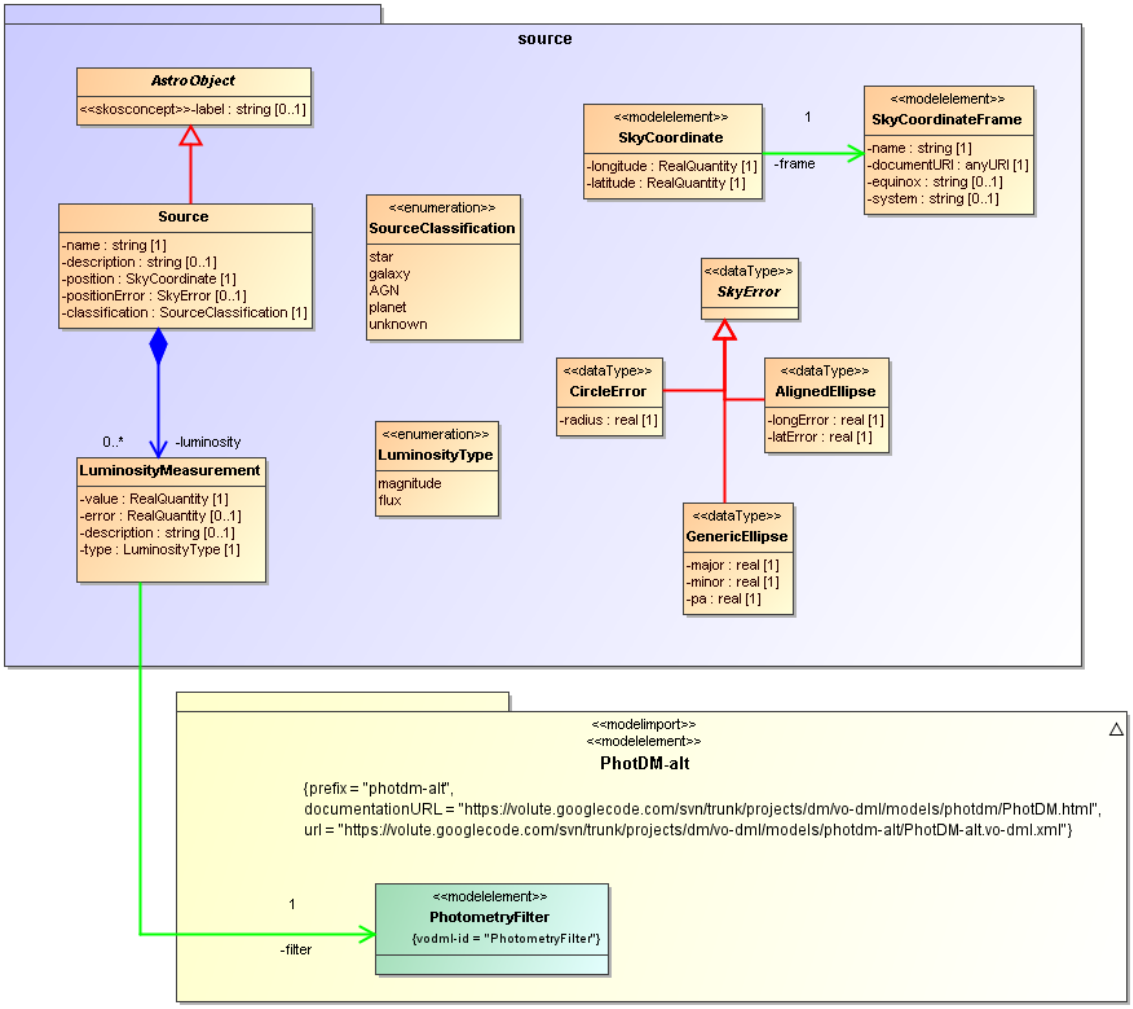


Figure 7 Simple Source data model used for illustrations in this document.

Appendix B Other modelling languages

The goal of VO-DML is to provide a domain specific modelling language, for the VO domain. Similar efforts have been made for other domains and a meta-meta-language has even been proposed to provide common semantics for such meta-languages. Here we shortly link to some of these efforts.

B.1 XMI

See [XMI

B.2 (E)MOF

…

B.3 EMF/Ecore

B.4 OData/CSDL: Common Schema Definition Language

Language to support OData protocol, standard web protocol for querying, retrieving, updating, deleting data on the web.

See http://www.odata.org/documentation/odata-v3-documentation/common-schema-definition-language-csdl/#18_Informative_XSD_for_CSDL

See also Gdata[15], "The Google Data Protocol is a REST-inspired technology for reading, writing, and modifying information on the web."

B.5 RDF Schema

http://www.w3.org/TR/rdf-schema/

[15] https://developers.google.com/gdata/