International

Virtual

Observatory

Alliance

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

Version 1.0-20140129

Working Draft 2014 January 29

This version:
0.x-20140129
Latest version:
0.x-20130510
Previous version(s):

Editors:
Gerard Lemson
Laurent Bourgès
Omar Laurino

Authors:
UTYPE-s tiger team+Laurent Bourges

Abstract

We propose that data models in the IVOA should be defined using a consistent and standardized modelling language, or meta-model. Having such a uniform language for all models allows these models to be used in a homogeneous manner and allows a consistent definition of reuse of one model by another. Moreover it allows us to add a consistent identification mechanism of components in the model so that these can be referenced in an explicit and

uniform manner. The latter is the requirement for annotating elements in data serializations with elements of a data model that they represent, often referred to as UTYPE-s in the IVOA.
In this WD we propose such a language, which we name **VO-DML** (VO Data Modelling Language). VO-DML is a conceptual modelling language that is agnostic of serializations or physical representations. This allows it to be designed to fit as many purposes as possible. VO-DML can be considered as a subset of UML, more precisely as a particular representation of a UML2 Profile [TBD ref]. From UML it only uses elements form its language for describing "Class Diagrams".
VO-DML has a representation as an XML dialect which we refer to as VO-DML/XML; indeed it was originally conceived as a simplified representation of UMLs XML serialization language, XMI. That language must express all of UML and hence is very verbose, implicit and hard to use. VO-DML is restricted to describing static data structures and VO-DML/XML aims to be concise, explicit and easy to use in code that needs to interpret annotated data sets. An earlier version of VO-DML has been used extensively in the Simulation Data Model effort[1], and greatly simplified the creation of derived representations from the core model.
Arguably the most important use case for VO-DML is the UTYPE specification
[Schematron] http://www.schematron.com/
[1]. [SIMDM] http://www.ivoa.net/documents/SimDM/index.html
[UTYPES] which uses it to provide a translational semantics for VOTable annotations. These annotations allow one to explicitly describe how instances of types from a data model are stored in the VOTable.
VO-DML as described in this document is an example of a domain specific modelling language, where the domain is here defined as the set of data and meta-data structures handled in the IVOA and Astronomy at large.
VO-DML provides a custom representation of such a language, which has however concepts in common with most other meta-model. In the appendixes we will provide some explicit links between the concepts.

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 Introduction Background and motivation

TBD new intro

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 describe 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-DML/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 [UTYPE] 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[2] 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-DML/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 latter is defined using an XML schema together with a Schematron[3] file which we denote together as **VO-DML/Schema**. The relation between VO-DML and VO-DML/XML is equivalent to the relation

[2] https://code.google.com/p/vo-urp/)
[3] http://www.schematron.com/

between a UML [5] (see also [10]) model and its representation as file in the XML serialization format XMI [9]. VO-DML is directly derived from UML, most of its modelling constructs have a counterpart in UML. One can in fact interpret VO-DML as a UML *Profile*[4] [10], a domain specific "dialect" of UML. For this reason we also provide a (non-normative) UML representation of VO-DML, which we named **VO-UML**, and which can assist in presenting a model in a form that is more easily interpreted, and is also the preferred format for "whiteboard" modelling. In Appendix A we give a table summarizing the relation between the VO-DML and UML concepts from which they are derived.
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 excluded from our language. Constraints are supported, but in a restricted manner. The titles of the subsections in section 3 directly refer to these VO-DML concepts. Inside of each section we describe the concept, show its definition in VO-DML/Schema and a VO-DML/XML example, and illustrate how one might express the element in VO-UML. Here we use the implementation of VO-UML in MagicDraw Community Edition 12.1[5].
VO-UML has been expressed at least in one UML modeling tool as a UML *profile*, using *stereotypes* with *tag definitions* to enable modeling of domain specific components in the graphical tool.
But note, though we use VO-UML for illustrations, and have used it to define some models, VO-DML is *not dependent* on UML or any tools for defining models in UML. VO-DML data models need not have a UML representation, but they MUST have a serialization in terms of VO-DML/XML (see section 4 for details on how to define data models). Hence the VO-UML part in this spec is INFORMATIVE, not NORMATIVE.

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 [6] and further constrained by an associated Schematron file, vo-dml.sch.xml[7]. 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.

[4] http://www.uml-diagrams.org/profile-diagrams.html
[5] TBD add text/a link to the profile.
[6] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.xsd
[7] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.sch.xml

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 VO-DML/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 is explicitly and formally defined using XML schema, whilst Schematron [Schematron] is used for adding further constraints that cannot be expressed in XML Schema alone[8]. 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 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 **Error! Reference source not found.**.

2.3 Other representations

In section Appendix Bwe discuss how one might represent VO-DML data models in application specific formats. These can be considered *physical models* that should implement a *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, an instantiation format tailored for VO-DML models. This is discussed in section B.4.

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 we give examples of a VO-DML/XML serialization. Where appropriate we also provide a representation of the concept

[8] We have chosen to use XML Schema as the core language for VO-DML/XML, as we feel it is more familiar to the IVOA community.

in VO-UML, and provide a link to the UML concept from which it is derived[9]. The VO-UML representation is graphical and we have used a particular UML modeling tool (MagicDraw Community Edition 12.1) to create the images. It may be that other tools produce a different representation, or are unable to reproduce some of the concepts[10]. TBC?
The different child components of the main concepts are described in subsections headed by corresponding titles. 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 D. Its VO-DML/XML representation can be found in the Volute project in GoogleCode[11] [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 defining VO-UML examples, we have used screenshots obtained from MagicDraw[12] Community Edition 12.1, which supports UML 2.0[13], serializing its models to XMI 2.1[14].

3.1 Model

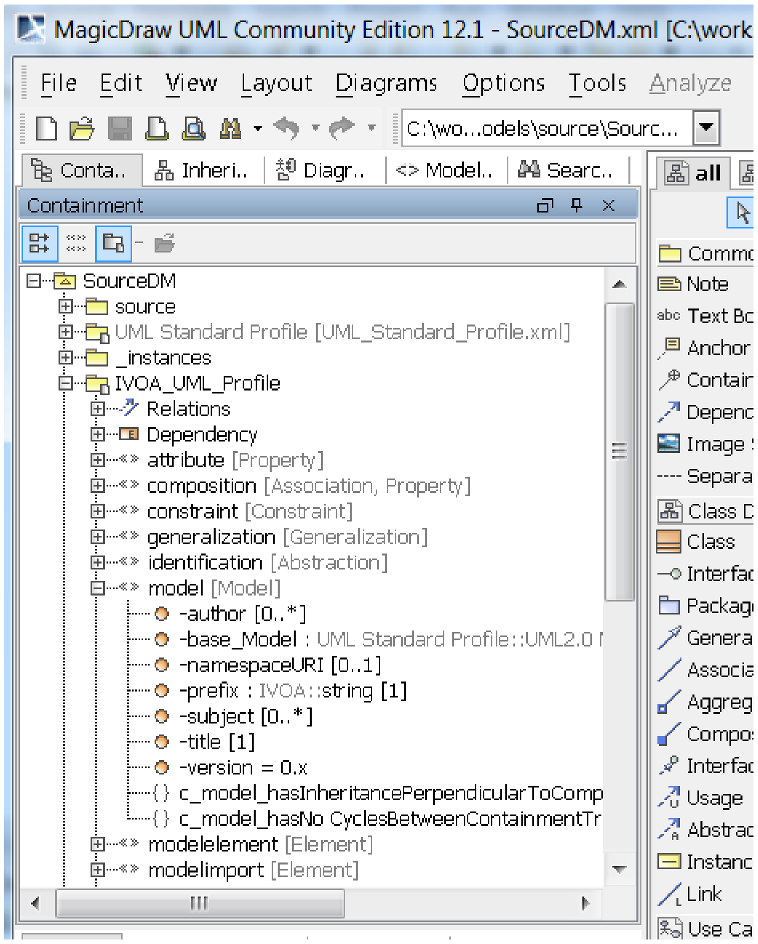
A (data) model represents a coherent set of type definitions, by which it represents the concepts that have an explicit place in its universe of discourse, i.e. which concepts one can talk/"discourse" about. Each data model is generally represented by a single document. **Model** is an explicit concept 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

[9] We use the very useful site http://www.uml-diagrams.org/ for the links. TBD whether they should go to formal specification.
[10] It should once more be made clear that the current spec *does not* mandate the use of some particular UML modelling tool. But a UML representation different from VO-UML MUST NOT be used to define an IVOA data model *without* an explicit mapping to VO-DML. It would in fact be a useful exercise to provide UML Profiles or something akin to that for representing VO-UML in more tools.
[11] https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/source/SourceDM.vo-dml.xml
[12] http://www.nomagic.com/products/magicdraw.html, edition CE 12.1 is no longer available online.
[13] http://schema.omg.org/spec/UML/2.0
[14] http://schema.omg.org/spec/XMI/2.1

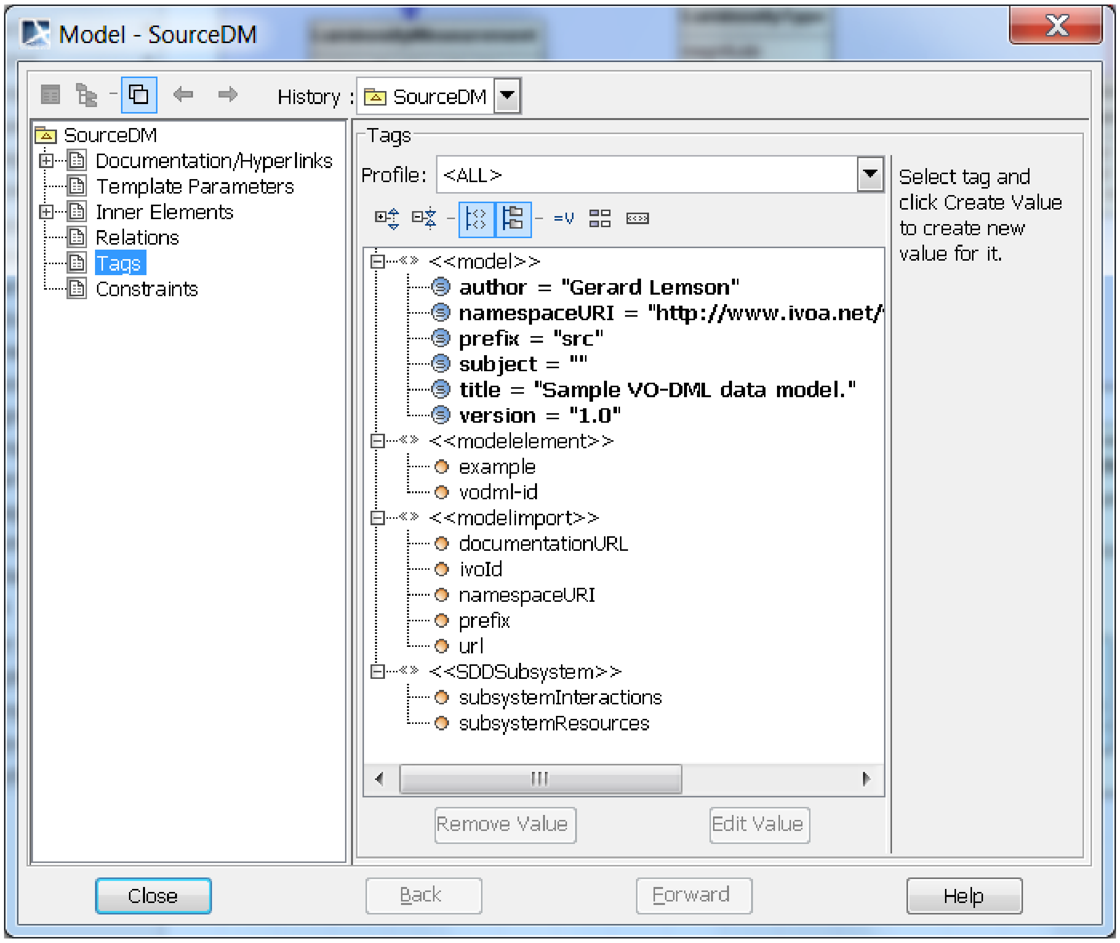
define extra metadata about the model and which correspond to the metadata elements defined in the subsections below.



```
SourceDM
    source
    UML Standard Profile [UML_Standard_Profile.xml]
    _instances
    IVOA_UML_Profile
        Relations
        Dependency
        attribute [Property]
        composition [Association, Property]
        constraint [Constraint]
        generalization [Generalization]
        identification [Abstraction]
        model [Model]
            -author [0..*]
            -base_Model : UML Standard Profile::UML2.0 I
            -namespaceURI [0..1]
            -prefix : IVOA::string [1]
            -subject [0..*]
            -title [1]
            -version = 0.x
            { } c_model_hasInheritancePerpendicularToComp
            { } c_model_hasNo CyclesBetweenContainmentTr
        modelelement [Element]
        modelimport [Element]
```

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:



```
SourceDM
  Documentation/Hyperlinks
  Template Parameters
  Inner Elements
  Relations
  Tags
  Constraints
```

Tags

Profile: <ALL>

Select tag and click Create Value to create new value for it.

```
<<model>>
    author = "Gerard Lemson"
    namespaceURI = "http://www.ivoa.net/
    prefix = "src"
    subject = ""
    title = "Sample VO-DML data model."
    version = "1.0"
<<modelelement>>
    example
    vodml-id
<<modelimport>>
    documentationURL
    ivoId
    namespaceURI
    prefix
    url
<<SDDSubsystem>>
    subsystemInteractions
    subsystemResources
```

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. Model is furthermore represented by the single root element defined in the schema, named `model`. This has type Model and has a uniqueness constraint defined on the vodml-id-s of all its contained elements.

```
<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>
  <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. The UTYPE specification uses this name as the prefix for all references into the model and hence certain constraints are put on its definition. Furthermore the name should be unique within the IVOA context[15]. The same rule applies also for references inside of the model, generally using the <utype> element. I.e. also when assigning for example a datatype from inside the current model to an attribute or other Role, the name of the model MUST be used as prefix.

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 author: string[0..*]

List of names of authors who have contributed to this model.

3.1.5 version : string [1]

Label indicating the version of this model.

3.1.6 previousVersion : anyURI [0..1]

URI identifying a VO-DML model that is the version from which the current version of model is derived.

3.1.7 lastModified: dateTime [1]

Timestamp when the last change to the current model was mad.

[15] Might this be renegotiated in on-going discussions and review process?

3.1.8 import : ModelProxy [0..*]

An 'import' element indicates a dependency on an external, predefined VO-DML data model. Types from that model may be used for referencing, extension and assignment to attributes. Types from the external model MUST NOT be used for composition relationships.

VO-DML/XML

```
<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/v
o-dml/models/photdm/PhotDM.html</documentationURL>
  </import>
...
```

3.1.9 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>
...
```

3.1.10 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”.

3.1.11 dataType : DataType [0..*]

Collection of DataType-s defined directly under the model.

3.1.12 enumeration : Enumeration [0..*]

Collection of Enumeration-s defined directly under the model.

3.1.13 primitiveType : PrimitiveType [0..*]

Collection of PrimitiveType-s defined directly under the model.

3.2 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.

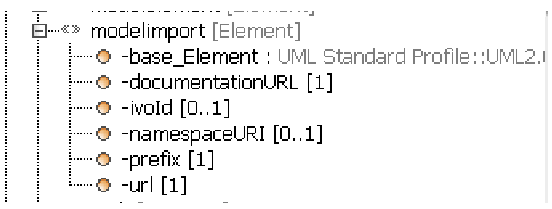


Figure 2 Stereotype definition for <<modelimport>>.

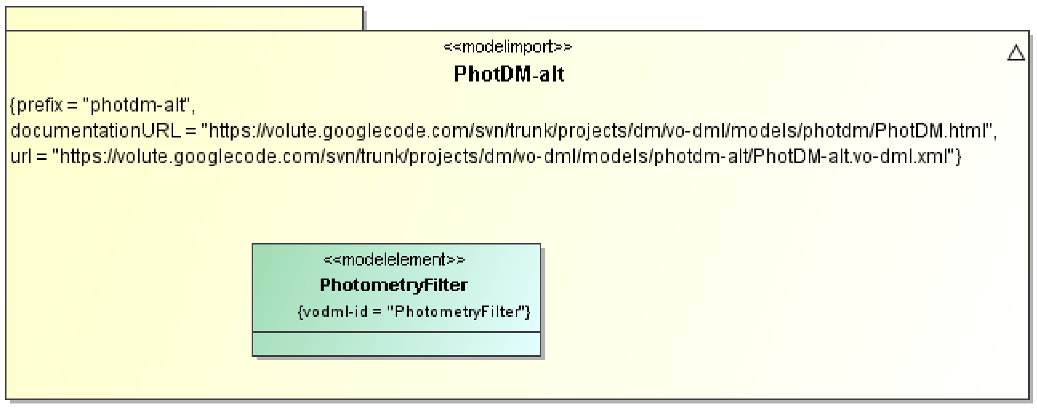

<<modelimport>>
PhotDM-alt
{prefix = "photdm-alt",
documentationURL = "https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/photdm/PhotDM.html",
url = "https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/photdm-alt/PhotDM-alt.vo-dml.xml"}
<<modelelement>>
PhotometryFilter
{vodml-id = "PhotometryFilter"}

Figure 3 Graphical representation of an imported model. Note the usage of the stereotype <<modelimport>> and the values assigned to the various tags. Also shown is a type imported with the model. It must have an explicit vodml-id assigned, which is accomplished using the <modelelement>> stereotype (see 3.3).

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>
```

3.2.1 name : string [1]

Name by which imported model is used in the current model and its documentation. This name MUST be the same as the 'name' given to the imported model in the VO-DML document from which it is imported. All utypes pointing to elements in the imported model MUST use this name as prefix.

3.2.2 ivoId : anyURI [0..1]

IVO Identifier of the imported model if that exists, i.e. if that has been registered in an IVOA Registry.

3.2.3 url : anyURL [1]

URL from which the imported VO-DML model document can be downloaded.

3.2.4 documentationURL : anyURI [1]

URL where a documentation HTML file for the remote model can be downloaded. This SHOULD be a document that contains anchors for each element that has as name attribute the vodml-id of that element. I.e. it is assumed that the vodml-id-s of the imported types can be added onto this documentationURL (should end with a #?) so that a direct link to the documentation for a referenced data model element can be found.

3.3 ReferencableElement [16]

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

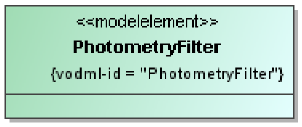
[16] TODO Arnold Rots argues this type should be renamed. To ReferrableElement or at least ReferenceableElement.

model, or from an external context. VO-DML/Schema defines the abstract type ReferencableElement that is a super type of all types representing such referencable concepts support. It contains an identifier element named <vodml-id> which must be unique within the model.

All referencable elements also have a *name*, and a description. The name MAY be used in standard approaches to deriving the vodml-id from the structure of the model. But this is not obligatory. The name must often be unique in many contexts in which a referencable element is defined. For example all types defined as direct children inside of a package must have a name which is unique in the context of that package. Similarly attributes must be unique in the definition of the type they are defined in, and this must even be true for the whole collection of roles in the inheritance hierarchy of the type.

VO-UML

This type is partially represented in VO-UML by the <<modelelement>> stereotype. That stereotype defines a tag 'vodml-id'. When assigning the stereotype to a particular model element allows one to define an explicit value fo the vodml-id of the element, rather than the default value that would be assigned by some UM modeling tool or could be generated by the VO-UML itself using a grammar like used in SimDM [SimDM] and described in Appendix Cbelow.
Note that in particular types partially defined on an imported model MUST be assigned the vodml-id they have in their model as defined by its formal VO-DML/XML representation.



VO-DML/Schema

All main modeling elements in the VO-DMl XSD (apart from `Model`) 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 expresses 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:element name="name" type="xsd:string" minOccurs="1"/>
    <xsd:element name="description" type="string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

VO-DML/XML
TODO

3.3.1 ReferencableElement.vodml-id : ElementID [1]
Identifier for the containing element. The type ElementID defines the syntax of the vodml-id as shown in the VO-DML/XSD snippet above.

3.3.2 ReferencableElement.name : string [1]
The name of the model element. May be restricted with uniqueness constraints in subclasses of ReferenceableElement.

3.3.3 ReferencableElement.description : string [0..1]
Human readable description of the model element. Note the multiplicity constraints. In principle every model element SHOULD have a meaningful description, but no tool will be able to check that a description is correct. Since a meaningless string can easily be provided if one wants to evade a possible not null constraint, we simply allow the description to be empty.

3.4 ElementRef
To refer to a ReferencableElement from inside a model, for example to indicate the data type of an attribute or other role, one must use an ElementRef. This contains a single element named <utype>, the value of which must be the vodml-id of the referenced element, prefixed by the name of the model the ReferencedElemtn belongs to. This can be the same model as the referencing element, or a model imported by that element's model. We use the name 'utype' as also the value of UTYPE-s in other IVOA data formats must use this same syntax [UTYPE].

VO-UML
This type is not explicitly represented in VO-UML. But any usage of one type by another, be it as dataty[pe of an attribute, as target type of a relation, will give rise to an ElementRef definition in the corresponding VO-DML/XML.

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

  <xsd:complexType name="ElementRef">
    <xsd:sequence>
      <xsd:element name="utype" type="UTYPE">
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
```

3.4.1 ElementRef.utype : UTYPE [1]

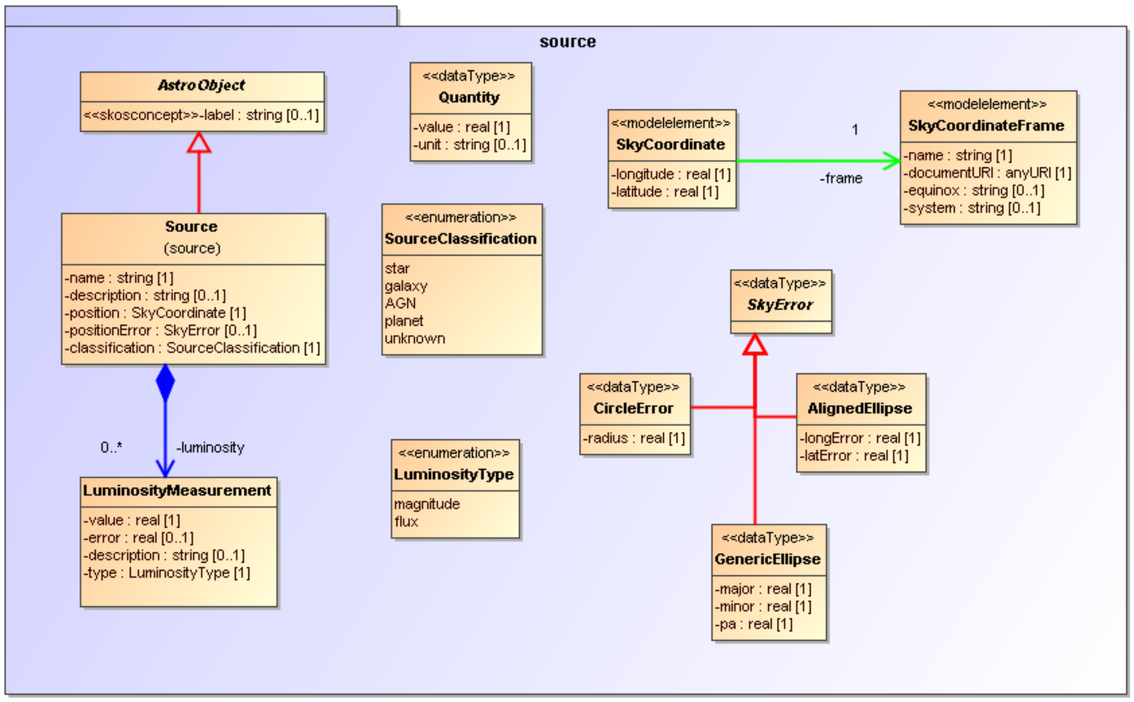
The element identifying the referenced target element. The syntax of the utype consists of the name of the model, a ':' and the vodml-id of the referenced element. In mock BNF:
<utype> :== <model-name> ':' <vodml-id>

3.5 Package extends ReferencableElement

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

In VO-UML a **Package** is represented by the UML *Package* element, shown in the diagram as the purple, tabbed rectangle. Types owned by the package may be shown inside the symbol, or the package name may be placed within parentheses below the name of the type. As shown by the Source type in the diagram.



VO-DML/Schema

In VO-DML/Schema, **Package** is represented by a complexType 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.

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

Collection of ObjectTypes defined in this package.

3.5.2 dataType : DataType[→] [0..*]

Collection of DataTypes defined in this package.

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

Collection of PrimitiveTypes defined in this package.

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

Collection of Enumerations defined in this package.

3.5.5 package : Package[→] [0..*]

Collection of child packages defined in this package.

3.6 Type extends ReferencableElement

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 super type of the declared type.

VO-DML/Schema

An abstract complexType named `Type` is introduced that is the super type 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 *super type*. The super type is identified by a `utype`.

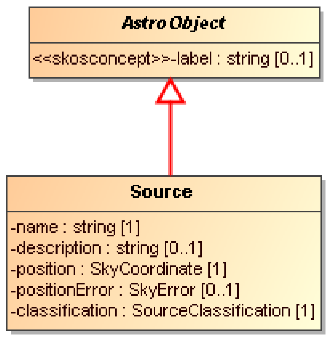
```
<xsd:complexType name="Type" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <xsd:sequence>
        <xsd:element name="extends" type="ElementRef" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="abstract" type="xsd:boolean"
         default="false"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

3.6.1 Inheritance

Indicates the typical “is a” relation between the sub-type and its super-type (the one pointed at). It implies that every object that is an instance of the subtype, is also an instance of the supertype. In VO-DML we do not support multiple inheritances. Furthermore ObjectTypes extend ObjectTypes, DataTypes extend DataTypes etc.
This concept corresponds to UML's Generalization. When comparing to UML

VO-UML

Inheritance relationship is represented by a UML generalization arrow from subtype to super type. The arrow head is an unfilled triangle.


AstroObject
<<skosconcept>>-label : string [0..1]
Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]

VO-DML/XML

```
    <objectType>
      <vodml-id>source/Source</vodml-id>
      <name>Source</name>
...
      <extends>
        <utype>src:source/AstroObject</utype>
      </extends>
...
```

3.7 ValueType extends 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>
```

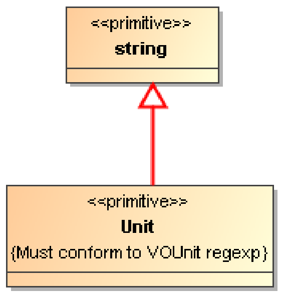
3.8 PrimitiveType extends ValueType

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

A primitive type is represented by the stereotype <<primitive>> and the name of the type. If it extends an existing type the *description*, defining the restriction, may be represented by text below the type name.



VO-DML/SCHEMA

```
  <xsd:complexType name="PrimitiveType">
    <xsd:complexContent>
      <xsd:extension base="ValueType">
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

VO-DML/XML

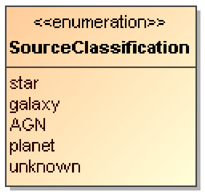
```
<primitiveType>
  <vodml-id>quantity/Unit</vodml-id>
  <name>Unit</name>
  <description>
    Must conform to definition of unit in VOUnit spec.
  </description>
  <extends>
    <utype>ivoa:string</utype>
  </extends>
</primitiveType>
```

3.9 Enumeration extends ValueType

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

VO-UML

An enumeration is represented by the stereotype <<enumeration>> and the name of the type. Below the name the literals are listed.



VO-DML/XSD

```
  <xsd:complexType name="Enumeration">
    <xsd:complexContent>
      <xsd:extension base="PrimitiveType">
        <xsd:sequence>
          <xsd:element name="literal" type="EnumLiteral"
               maxOccurs="unbounded">
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

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

3.9.1 Literal

A Literal is a possible value of an enumerated type. It is a ReferencableElement and therefore has a *vodml-id* and a *name* which represents the value, and can be annotated with a *description*.

VO-DML/XSD

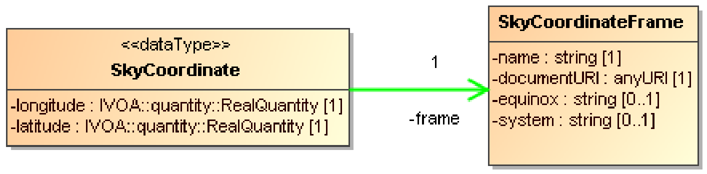
```
  <xsd:complexType name="EnumLiteral">
    <xsd:complexContent>
      <xsd:extension base="ReferencableElement">
        <xsd:sequence>
         </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

3.10 DataType extends ValueType

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. As is the case for the other ValueTypes, 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 datawith respect to which the rest of the value should be interpreted. For example SkyCoordinate may have a reference to a SkyCoordinateFrame to help interpret the values of the longitude/latitude attributes.
TODO justify making this distinction.

VO-UML

This concept is directly derived for UML's Data Type[17]. 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>
```

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

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>
    <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>
```

3.10.1 attribute: Attribute[→] [0..*]

Collection of Attribute definitions.

3.10.2 reference: Reference[→] [0..*]

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

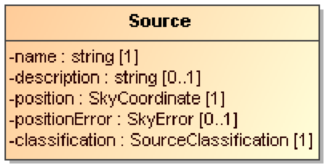
3.11 ObjectType extends ValueType

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[18]. 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 [TBD defend/explain usage of self-evident here and above]. 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 "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..


Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]

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>
```

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

```
    <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>
...
```

3.11.1 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?]

3.11.2 attribute: Attribute[→] [0..*]

Collection of Attribute definitions.

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

Collection of collection definitions.

3.11.4 reference: Reference[→] [0..*]

Collection of Reference definitions.

3.12 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 super type 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>
```

3.12.1 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`.

3.12.2 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.

3.12.3 subsets: ElementRef

The *subsets* element indicates that the Role redefines a Role defined on a super type. It can only do so by restricting the *datatype* of the Role to a subtype of the *datatype* of the Role it overrides. This is a common design pattern and directly borrowed from UML.

VO-UML

TBD

3.13 Attribute extends Role

An **Attribute** is the role a **ValueType** 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.
Attribute restricts the possible types of the **Role**'s datatype attribute to **ValueType**-s only.

VO-UML

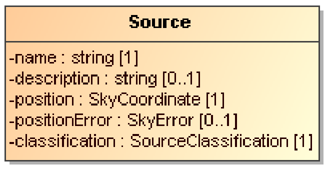

Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]

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

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

3.13.1 constraints : Constraints [0..1]

Attributes can be constrained in a number of simple ways that are defined by the Constraints type. Assigning such a Constraint to the attribute indicates restrictions on the value of the attribute.

3.13.2 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.

3.14 Constraints

It is useful to be able to attach constraints to data model elements beyond the multiplicity. Constraints apply to instances of types or roles. In general these can be complex and might require a language such as OCL (=Object Constraint Language [TBD add reference]). In VO-DML we pre-define some simple constraints, and only attach them to **Attribute** definitions.

3.14.1 minLength : integer

For a string attribute, the minimum length the string can have.

3.14.2 maxLength : integer

For a string attribute, the maximum length the string can have. A value <=0, indicates the string has no length limit.

3.14.3 length : integer

For a string attribute, the exact length the string must have.

3.14.4 uniqueGlobally: boolean

If the value of this field is set to **true**, this indicates that the value of the attribute to which the constraint is applied, must be "globally" unique. I.e. when applied to an attribute, this indicates that in the collection of all[19] instances of the type owning the attribute, the attributes must have unique values.

3.14.5 uniqueInCollection : boolean

If the value of this field is set to true,, this indicates that the value of the attribute must be unique in the direct collection of objects that the parent type belongs to.

3.14.6 complexConstraint: ConstraintExpression

An expression constraining the value to which the constraint is applied. May be human readable, could become a regular expression, or maybe OCL expression in future. To be implemented by hand in target representations of the model.

[19] This leaves the context open. It could be a database with objects, or the IVOA, or the "world". Generally to be defined by the application context for which the model is to be used.

3.15 ConstraintExpression

Sometimes constraints are more complex than some simple limit on ranges or values supported by the various explicit choices in the Constraints type. Designers can define more complex expressions using the current class.

3.15.1 expression: string

This attribute holds the expression defining the constraint.

3.15.2 language: string

This attributes defines the language in which the constraint in **expression** is expressed. This attribute is enumerated with the following values

- **XSD** expression is an XSD 'pattern' regular expression [TBD ref]
- **Java** expression is a java.util.regex.Pattern regular expression [TBD ref].
- **OCL** expression is an OCL constraint [TBD ref]
- **Custom**: expression is natural language sentence, to be interpreted and implemented by humans.

3.16 *Relation* extends *Role*

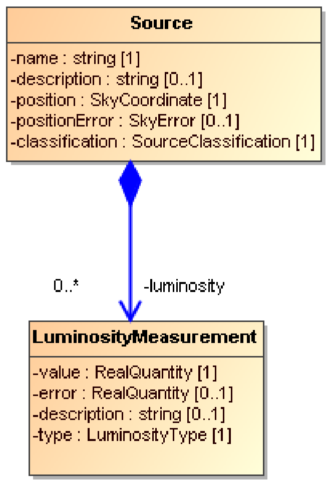
A **Relation** is a role played by an **ObjectType** in the definition of another ObjectType, or possibly a DataType. It indicates that the ObjectType that is the target (datatype) of the relation is related in some fashion to the source type.
In VO-UML relations are indicated by lines connecting the two types in the relation, always with an arrow on the side of the type that plays the role. Details depend on the type of relation. VO-DML defines two types, **Collection** and **Reference** that are both subtypes of Relation.
Note, also the inheritance relation would seem

3.17 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. In VO-DML we impose the restriction that an ObjectType can only be the target of at most one Collection relation. This includes relationships inherited from a super type. The only case where this rule may be broken is where a collection explicitly *subsets* another, inherited collection.

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.


Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]
0..*
-luminosity
LuminosityMeasurement
-value : RealQuantity [1]
-error : RealQuantity [0..1]
-description : string [0..1]
-type : LuminosityType [1]

VO-DML/Schema

```
  <xsd:complexType name="Collection">
    <xsd:complexContent>
      <xsd:extension base="Relation">
        <xsd:sequence>
          <xsd:element name="isOrdered" type="xsd:boolean"
               default="false" minOccurs="0">
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

VO-DML/XML

```
    <objectType>
      <vodml-id>source.Source</vodml-id>
      <name>Source</name>
...
      <collection>
        <vodml-id>source.Source.luminosity</vodml-id>
        <name>luminosity</name>
        <description>
          Collection of luminosity measurements for the parent source.
        </description>
        <datatype>
          <utype>src:source.LuminosityMeasurement</utype>
        </datatype>
        <multiplicity>
          <minOccurs>0</minOccurs>
          <maxOccurs>-1</maxOccurs>
        </multiplicity>
      </collection>
    </objectType>
```

3.18 Reference extends Relation

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*.

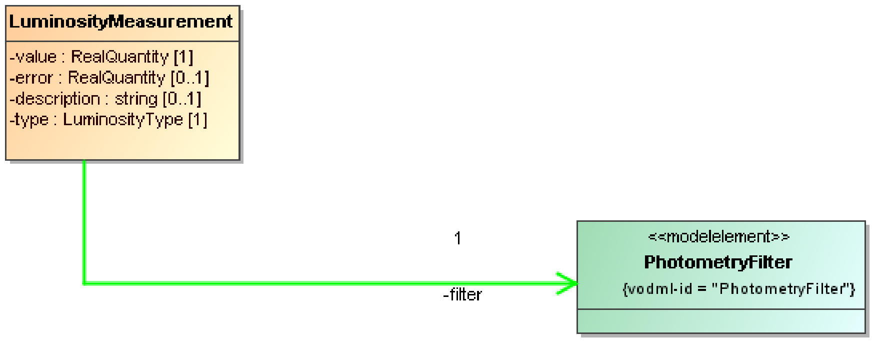

LuminosityMeasurement
-value : RealQuantity [1]
-error : RealQuantity [0..1]
-description : string [0..1]
-type : LuminosityType [1]
1
-filter
<<modelelement>>
PhotometryFilter
{vodml-id = "PhotometryFilter"}

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

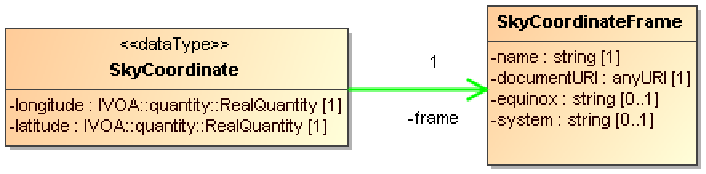


Figure 6 Reference from a DataType to an ObjectType

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>
```

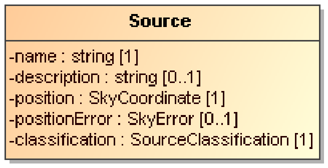
```
. . .
```

3.19 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 former 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 '*'.[20]

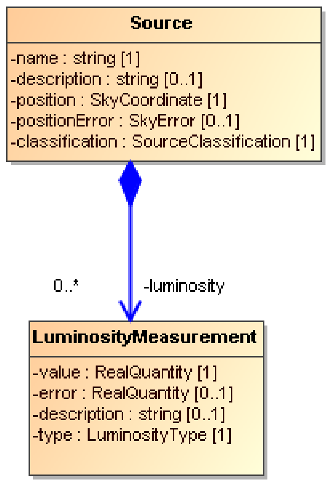
VO-UML

In VO-UML the multiplicity, 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 two dots, '..'. The value of -1 for maxOccurs is represented by a '*':


Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]

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

[20] TBD We may consider restricting 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.I.e. they would work more as array declarations in many languages, for which the initial length must be set. Motivation comes mainly from relational mapping that would be complicated . Attributes generally map to a column,, and unless one designs a special way to store multiple values in a single column, something which breaks 1st normal form, one may need a special child table for storing the values. With a fixed length array one may decide that each element in the array gets its own column.


Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-positionError : SkyError [0..1]
-classification : SourceClassification [1]
0..*
-luminosity
LuminosityMeasurement
-value : RealQuantity [1]
-error : RealQuantity [0..1]
-description : string [0..1]
-type : LuminosityType [1]

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 Rules on defining data models in the IVOA

Assuming one has a standardized data modeling language now allows one to be more formal on what the acceptable end product of an IVOA data modeling effort should be. Here we define the procedure for creating an IVOA data model and related resources according to the VO-DML philosophy.
Before stating the formal rules, first some slightly philosophical considerations

- Decide *why* one wants to create a data model. Default goal for any IVOA data model should be that it allows existing and future databases to describe their contents (at least partially) in terms of a common model, often called a *global schema*. In certain cases a model may be developed to provide support for a particular application area, for example when defining a data access protocol. Here one can decide to support faithful serialization of data models in targeted XML documents as well as annotated serialization in VOTable.
- Decide on universe of discourse: what concepts must be described? How rich should the model be?
- Create a conceptual/logical model. In drawings on whiteboard ("VO-UML"), then transcribe to 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, if possible automated derivation methods of VO-DML to target representation.

4.1 Procedure

A new data model MUST have a VO-DML/XML representation. This is an XML document that MUST be valid with respect to the vo-dml.xsd schema[21] and the rules embodied by the vo-dml.sch.xml[22] Schematron file. How this representation is produced is not important. It may be written by hand, it may be derived from a UML/XMI representation using an XSLT script, 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

[21] The schema is currently available under http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.xsd .
[22] 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.

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. Of particular importance is the model named "ivoa" which SHOULD be imported by all IVOA data models, the IVOA_Profile[23]. This model contains a set of predefined data types, mainly primitive types, such as string, boolean, integer and real. It also predefines a set of Quantity types that the UTYPEs spec uses to allow mapping of scientific measurement values directly to FIELD-s and PARAM-s in a VOTable. These standard types SHOULD be used to represent these common concepts in all models.
A data model MAY produce an XML schema that represents a faithful representation[24] of the model. If this is done, the schema MUST allow one to define XML documents representing instances of the data model in 1-1 mapping. If the IVOA at some point defines one or more standard mappings between VO-DML and XSD, one of those mappings MUST be used (see B.1 for a first approach to such a mapping). The XSD elements MUST (where appropriate) contain an app-info element identifying the model element that is represented using its utype (format of app-info is TBD).
A data model MAY[25] produce a TAP schema matching the data model. If this is done, the TAP schema MUST represent a 1-1 relational mapping of the data model. If the IVOA at some point defines one or more standard mappings between VO-DML and TAP_SCHEMA, one of those mappings MUST be used (see B.2 for a first approach to such a mapping). The TAP_SCHEMA MUST be represented by a VOTable that has a TABLE element for each table in the schema and these MUST be annotated in the manner described in the UTYPE document with model elements identifying the corresponding VO-DML elements.

5 References

[2]. [Schematron] http://www.schematron.com/
[3]. [SIMDM] http://www.ivoa.net/documents/SimDM/index.html
[4]. [UTYPES] utypes mapping document
[5]. [UML] http://www.omg.org/spec/UML
[6]. [UML-Infr2.0] http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/

[23] 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 .
[24] See Appendix B for a definition of this concept. [TBD maybe already in introduction?]
[25] 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.

[7]. [UML-Sup2.0] http://www.omg.org/spec/UML/2.0/Superstructure/PDF/
[8]. [VO-URP] http://vo-urp.googlecode.com/
[9]. [XMI2.1] http://www.omg.org/spec/XMI/2.1/
[10]. http://www.uml-diagrams.org/

Appendix A Relation to UML

VO-DML is clearly strongly influenced by UML. In fact most of the modelling concepts form VO-DML have a counterpart in UML's class diagrams. We "only" add some features to some of the concepts, rename and possibly slightly re-interpret others, but especially severely restrict the UML concepts we use.
The following table lists the VO-DML concept, and provides the corresponding UML concept[26] with a short comment. The UML concepts are linked to a definition and more detailed description in the useful web site www.uml-diagrams.org.

VO-DML concept	UML concept	Comment
Model	Class Diagram	This is a loose relation, as UML does not have an explicit Model element. But each Model that we have drawn in VO-UML has used a class diagram.
ModelImport	none	UML models can not explicitly "import", "reuse" another model. UML has the concept of a package import, but that is more like Java's 'import' of a package, which allows one to refer to its classes without requiring a fully qualified name.
Package	Package	Essentially the same concept, providing a namespace for its contained elements, so that these need only worry about uniqueness of names inside the package context.
Type	Classifier	UML Classifier is a base class of all kinds of types. UML also has a Type, a supertype of Classifier. We use Classifier here as it is the nearest supertype to Class and DataType.
ObjectType	Class	Using a different name to avoid the software feel of the word Class.
ValueType	none	In VO-DML, ValueType is an abstraction of DataType, PrimitiveType and Enumeration, and is used by the definition of Attributes. In UML this role is in a sense taken by DataType, though without the restrictions on attributes!.
DataType	DataType	Possibly DataType is the real representative

[26] TBD which version of UML we will use.

		of our ValueType, though it allows the attributes. In that interpretation a PrimitiveType is a special datatype, namely one representing a single value, rather than a set of values, identified by a named attribute. This is certainly a valid interpretation.
PrimitiveType	<<primitive>> DataType	UML considers a PrimitiveType to be a special DataType, representing atomic values and without attributes. Note that the stereotype <<primitive>> is special to the UML tool we used in the diagrams in this document. Other tools may use different names for this, or may even not support primitive types altogether. [TBD depends on UML type?]
Enumeration	Enumeration	
Role	Structural Feature, Typed Element	Our name "Role" is derived from one view, "the role a type plays" in the definition of another type. UML refers to this very general concept as a Typed Element. In particular our Role-s are structural features though.
Attribute	(UML 1.x) Attribute, (UML 2.x) Property with a value type as datatype.	An Attribute is a Property with a value type as datatype. UML represents these as strings inside the class/datatype definition. Older versions of UML (e.g. 1.4.2) has an explicit Attribute concept.
Relation	(Binary) Association (End)	
Reference	Navigable Association End of shared binary association	
Collection	Composition Aggregation	
Inheritance	Generalization	We only allow single inheritance (like Java, C#, unlike C++ and UML).
Multiplicity	Multiplicity	Our representation in VO-DML/XML is closer to XML schema, with separate definitions of minOccurs and maxOccurs. But the same concept is intended that in UML is represented by an **m..n** notation

Appendix B Mapping to serialization meta-models

Here we give examples how one might map a VO-DML data model to a physical representation format described by its own meta-model. These mapping aim to be as close to 1-1 as possible, providing what we call faithful representations of the model. Developing standard mappings like these could greatly simplify a modeling process, as one need only define the conceptual model itself and use automated procedures to derive the serialization formats.

B.1 XSD

VO-DML concept	XSD concept
ObjectType	complexType
DataType	complexType
Enumeration	simpleType with restriction list elements for the EnumLiterals
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 complexType, type must be able to perform remote referencing
Extend	xsd:extension of type definition

B.2 RDB

[Follow typical Object-Relational mapping rules.

VO-DML concept	RDB concept
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 from child to parent
Reference	Foreign key form referrer to referent.

B.3 Java

[TBD]

B.4 VO-DML/I: A default serialization language?

To best express example instances of a VO-DML data model, it may be useful to define a 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.
[TBD Is it possible to integrate VO-DML/I into VO-DML/Schema? (Suggestion from Norman)]

Appendix C vodml-id generation rules

The only requirement on the <vodml-id> identifying model elements is that it is unique within the context of the model. In the past[27] rules have been defined for generating such unique values from the model that also provide a human readable representation of the location of the identified element in the model. Here we reproduce this generational grammar with some small changes. Note that uniqueness depends on rules on the uniqueness of names in a particular context, here represented by a location in a hierarchy:

```
vodml-id := [package-vodml-id | class-vodml-id | attribute-vodml-id |
         collection-vodml-id | reference-vodml-id | container-vodml-id
package-vodml-id := <package-name> [“.” <package-name>]*
class-vodml-id := package-utype “.” <class-name>
attribute-vodml-id := class-vodml-id “.” <attribute-name>
collection-vodml-id := class-vodml-id “.” <collection-name>
reference-vodml-id := class-vodml-id “.” <reference-name>
container-vodml-id := class-vodml-id “.” “CONTAINER”
```

Appendix D 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 .

[27] The Simulation Data Model [LINK] provided such a grammar for the first time for the list of utypes accompanying that model, which was taken over by an early draft of a UTYPE document [LINK].

The following figure shows a UML version of the model following the graphical rules explained in section 3 .

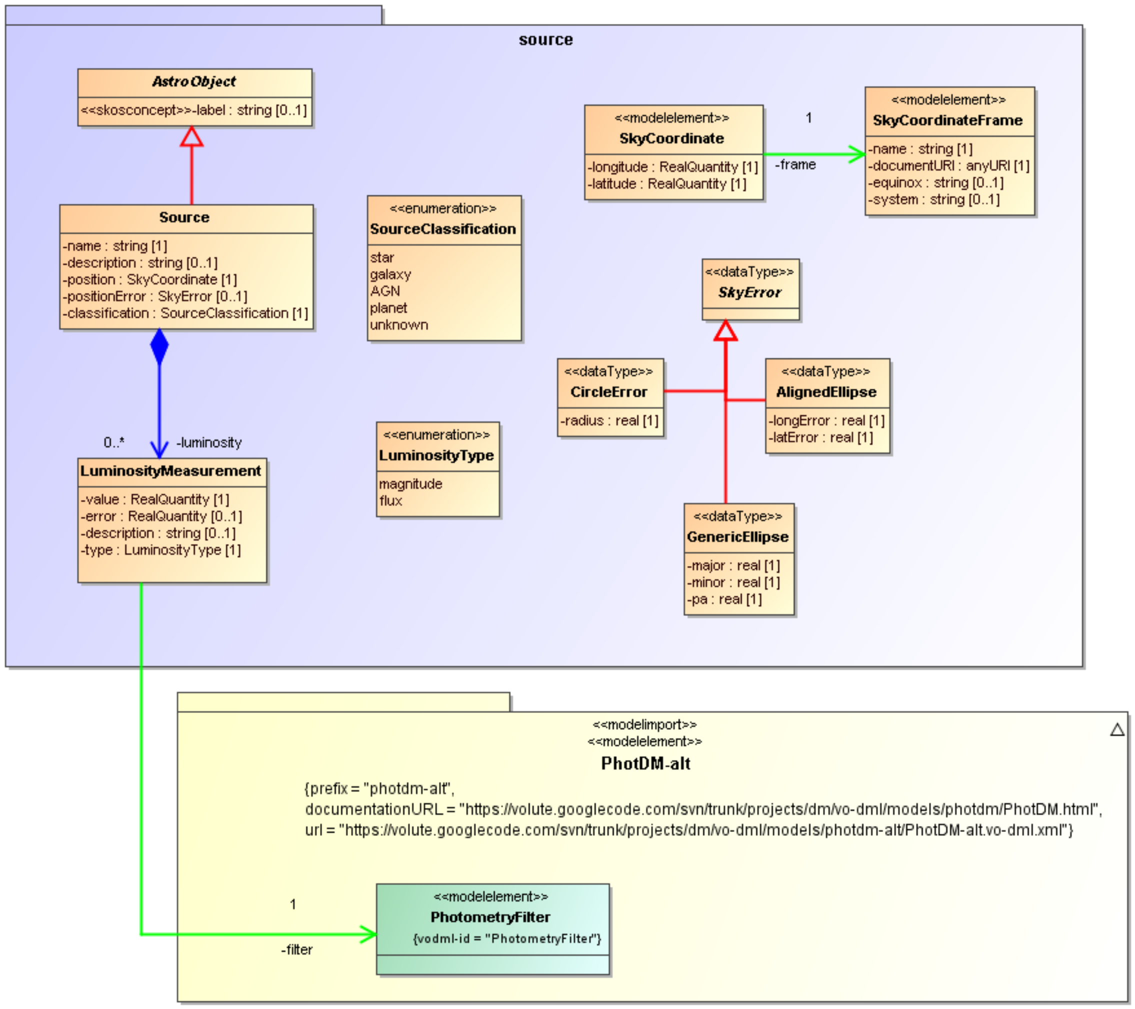


Figure 7 Simple Source data model used for illustrations in this document.