
*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*

!!! UNDER CONSTRUCTION !!!

TODO

- Decide on whether 3, 4 and appendix A can be merged.
- Executive summary.
-

**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 note we propose such a language, which we name VO-DML (VO Data Modelling Language). We discuss the requirements we feel any such a language should obey and describe their implementation in VO-DML. 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. One of the main requirements leading to an update of that version to the one proposed here is its use in the UTYPE specification. We describe how VO-DML helps in that effort.

## Status of This Document

This is an IVOA Note for the IVOA DM working group. The first release of this document was 2013-TBD.

*This is an IVOA Note expressing suggestions from and opinions of the authors. It is intended to share best practices, possible approaches, or other perspectives on interoperability with the Virtual Observatory. It should not be referenced or otherwise interpreted as a standard specification.*

*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 Motivation for and overview of VO-DML

- Data model in information integration.
- VO-DML as IVOA's Esperanto.
- Data model defines the universe of discourse expressed in common language.
-

Consistent language assists in understanding, simplifies modeling, avoid redundancy, allows reuse, defines targets for mapping (UTYPE-s), can be mapped consistently to various other representations, is designed for data modeling, enables interoperability, …

History: Simulation datamodel. UML (standard, implementation neutral), mapping to RDB (storage, TAP/ADQL compatible), mapping to XML schema (messaging, ingestion), Java (implementation of database management and web site). HTML (documentation), UTYPE-s (mapping).

# 2 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)
- Decide on universe of discourse: what must be described? How rich should model be?

- Create conceptual/logical model. In drawings on whiteboard, then transcribe to VO-DML. Define concepts completely, realizing that applications may pick and choose and transform. This is the common language
- Sometimes, in application contexts: derive one or more physical representations. Use as much as possible standard derivation of VO-DML to target representation.

## 2.1 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[2] and the rules embodied by the vo-dml.sch.xml[3] 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[4]. 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 app-info element identifying the model element that is represented using its utype (format of app-info is TBD).

---

[2] The schema is currently available under http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/xsd/vo-dml.xsd .
[3] 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.
[4] 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 .

A data model MAY[5] 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.]

# 3 Modeling Concepts

Here we list the main concepts in the VO-DML meta-model. VO-DML follows an object-oriented approach, but restricts itself to structure mainly. Operations are explicitly excluded from our language.
Data models in the IVOA have a particular goal, to facilitate interoperability, in particular to provide a common language to interpret, understand data sets of various forms. These are generally distributed and out of central control. They may be willing to send their data over the net in serialized form, but will generally not have operations to manipulate their data sets. Operations are gwenerally also very much application specific. Hence we feel we should concentrate on the data itself.

## 3.1 Model

A (data) model represents a coherent set of type definitions. It defines which concepts it is worth talking/"discoursing" about.

### 3.1.1 Package

A package provides "name spacing" to a data model. Packages divide the set of types in a model in subsets. Certain constraints, such as uniqueness of the type's name, are defined only within the context of the package. This concept is equivalent to similar concept in UML and other modeling systems and languages. A package groups related elements such as type definitions and possibly sub packages. Packages can depend on each other, which means that elements in one package can use elements in the target package in their definition. This relation is transitive. A package is similar to an XML namespace or a Java package.

### 3.1.2 ModelImport

A model can use the types defined in another model. To this end it must use an explicit import. The types in the remote model can be used in role definitions

[5] 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.

## 3.2 Type

The goal of a (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.
The most important categorization of Types is that between so called object types and value types.
[TBD find equivalent categorizations: reference type vs value type, ...]

### 3.2.1 Value Type

A ValueType represents a simple concept that is used to describe/define more complex concepts such as ObjectTypes. Instances of ValueType-s, i.e. *values*, are, in contrast to ObjectTypes not 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. This is in contrast to the case of ObjectTypes discussed below.

#### 3.2.1.1 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).

#### 3.2.1.2 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.

#### 3.2.1.3 DataType

A DataType is a value type with structure. The structure is generally defined by attributes on the DataType. An instance of a DataType, also a value, consists of giving values for the attributes. This is similar to ObjectTypes defined below, but in contrast to ObjectTypes, datatypes have no explicit identity, also DataType-s are defined by their value only. 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).

### 3.2.2 OjectType

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[6]. 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 "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.

## 3.3 Role

Types can "play a role" in the definition of another type. This concept is made explicit in VO-DML using the abstract base class *Role*. All explicit roles are defined in the subsections. A Role definition always indicates, through its *datatype* which type it is that is playing the role. Different *Roles* have different restrictions on which "type of" type can play the role

### 3.3.1 Attribute

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 such as age, mass, length etc.

### 3.3.2 Collection

An ObjectType may be "composed of" other object types. It does so through the definition of collections of the child types.
The life cycles of the child objects are governed by that of the parent.
In UML a composition relation is represented by a binary association end.

### 3.3.3 Reference

A reference is a relation that indicates a kind of *usage*, or *dependency* of one object on another. It is in general shared, i.e. many objects may reference a single other object. Accordingly the referenced object is independent of the "referee".

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".

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

### 3.3.4 Extends, inheritance

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

# 4 VO-DML/XML

Here we document the formal definition of the VO-DML meta-model. The official format of a VO-DML data model is an XML document conforming to the XML schema vo-dml.xsd[7] and further constrained by an associated Schematron file, vo-dml.sch.xml[8]. We will refer to this format as VO-DML/XML. The schema should be self-documented as much as possible.

In the following subsections the main model elements are described, and the corresponding XML schema element is given. For concrete cases a UML example is given with its mapping to VO-DML/XML [TBD do next as well? and an example instance is given using an XML element from a typical mapping of VO-DML to XML schema.] The examples are all extracted from the TAP data model that is described in Appendix B.

[TBD I am using a pinkish shading for concepts that I think are not really required or should be discussed]

## 4.1 *ReferencableElement*

All elements listed explicitly in this section are subclasses of *ReferencabbleElement*. This abstract base class has an identifier (containing the "utype") which must be unique in the model. This means that this element (well its concrete sub types) **can be referenced** by using their identifier. This is used explicitly in the ElementRef type that implements such references inside VO-DML. It can also be used in other contexts such as the VOTable's utype attributes [TBD add reference to the UTYPEs document]

*ReferencableElement* may also itself be identified with elements in other models using possible Identification relationships.

```
<xsd:complexType name="ReferencableElement" abstract="true">
  <xsd:sequence>
    <xsd:element name="identifier" type="Identifier"
      minOccurs="1"/>
    <xsd:element name="identifaction" type="ConceptIdentification"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

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

### 4.1.1 identifier : Identifier[→] [1]

The identifier contains a utype child element that must be unique within a model. *If* it is decided that each model must have a unique utype and that that utype should act as prefix for the utypes within the model, global uniqueness is ensure.

### 4.1.2 identification : Identification[→] [0..*]

Indicates that the concept represented by the element is similar to/can be identified with a concept defined in another model. That model must be imported by the current model, and the corresponding concept must have been imported as well.

## 4.2 Identifier

We use a separate Identifier class to model how one can identify elements in a data model. We do this to enable generalization of the single ID or utype. If this is deemed *not* useful it can be replaces with adding an @id attribute and utype element on ReferencableElement directly.

```
<xsd:complexType name="Identifier">
  <xsd:sequence>
    <xsd:element name="utype" type="UTYPE"/>
    <xsd:element name="altid" type="ExtIdentifier"
        minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="optional"/>
</xsd:complexType>
```

### 4.2.1 utype : UTYPE[→] [1]

The utype identifies the containing ReferencableElement and can be used to reference it from inside the model or from the outside, for example through a @utype attribute in a VOTable document.

### 4.2.2 Altid : ExtIdentifier[→] [0..*]

Any set of alternative identifiers for the parent concept RerferencableElement. E.g. the XMI:ID of an original UML model, or a publisherDID or ...

### 4.2.3 @id : xsd:ID [0..1]

An id attribute can be added to an identifier element. This is meant only to have meaning inside the document, for example it can be used for referencing the identified element. The XSLT script xmi2vo-dml.xsl uses this attribute as a place holder when generating the VO-DMl/XML representation form a UML diagram. In the first step the @id and utype will be set to the original xmiid attribute of the corresponding UML element. In the post-processing step where the utype-s are generated, the equality of @id and utype is used as criterion to decide that the utype should be generated according to the grammar. The two will not be the same if the utype was explicitly defined in the model itself. IN principle this role could be taken over by the altid element as well, currently there is some redundancy. [TBD this should be resolved].

### 4.2.4 Examples

#### 4.2.4.1 XML instance

```
<identifier id=”_ueykdahxeoimoiuseoi”>
  <utype>TAP:Column.name</utype>
</identifier>
```

## 4.3 UTYPE extends xsd:string

We introduce a special (simple)type for representing UTYPE values in identifier elements.. This is for the case where we want to put restrictions on the utype syntax. We can leave the syntax of utype-s free, or insist (only) on a prefix that MUST be the {UTYPE of the Model + ‘:’}, or insist on some grammar for deriving the utype from the data model element it identifies.
The HTML documentation SHOULD support looking up an element by UTYPE by providing an <a name=”utype”> anchor for each referencable element.

```
<xsd:simpleType name="UTYPE">
  <xsd:restriction base="xsd:string">
      <!-- TBD add a restriction, e.g. a format -->
  </xsd:restriction>
</xsd:simpleType>
```

## 4.4 ExtIdentifier

Alternative identifier possibly obtained from a source document from which the model was derived. May have a URL to the source and a simple string as id.

### 4.4.1 source: anyURI [0..1]

This element allows one to indicate the context within which the id element has a meaning, "where" one should go to look up the element using the id.

### 4.4.2 id: string [1]

The identifier of this element in the external context identified by the source
Modeled as a string, could be "anyType".

## 4.5 ElementRef

When referring to other elements in the current or a remote, imported model, an instance of ElementRef is to be used. It encapsulates the data structure allowing one to identify the referenced element.

```
<xsd:complexType name="ElementRef">
  <xsd:sequence>
    <xsd:element name="modelUtypeRef" type="xsd:string"
         minOccurs="0"/>
    <xsd:element name="utyperef" type="UTYPE" minOccurs="1"/>
  </xsd:sequence>
```

```
  <xsd:attribute name="external" type="xsd:boolean" use="optional"
       default="false"/>
  <xsd:attribute name="idref" type="xsd:string" use="optional">
</xsd:complexType>
```

### 4.5.1 modelUtypeRef : UTYPE [0..1]

Element holding on to the "utype" of the model to which the referenced element belongs. This should be the same as the utype of the current model, or of one of the imported models. If this element is not specified, it is assumed the referenced element belongs to the current model. Note, this is an attempt at normalizing the utype referencing model. The alternative is to use a prefix on the utyperef element to identify the model.

### 4.5.2 utyperef : UTYPE [1]

This element represents the utype of the element that is actually referenced. Depending on context there may be additional constraints on the type of the referenced element.

[TBD

Currently it is assumed that the utyperef string must have a prefix that identifies the model the referenced element belongs to. But see previous item. It is TBD whether different usages of a model might refer to a model's elements with different prefixes. If that were allowed, it would make it easier to support the use of custom models even before these have been accepted as a standard (if they ever will). In that case though we MUST have rules how to look up remote elements. E.g. substitute remote model's utype for prefix before looking up element. Or utype-identifiers in a model MUST not have a prefix?

]

## 4.6 ConceptIdentification

Represents a similarity/equivalence/... of a concept in an external data model or other set of concept definitions to the current element. We *do not* assume that the external model is VO-DML or imported. Hence the remoteid may be a utype or not. In principle the description could give that info. Or a remoteURL attribute should be added.

## 4.7 Model extends *ReferencableElement* [→]

This class represents the data model as a whole. It defines the type for the only possible root element in a VO-DML/XML document. It contains some meta-data elements and otherwise mainly type definition, possibly distributed over packages. A Model MAY import other VO-DML models and may also have still looser relations to other types of models.

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

```
        <xsd:element name="name" type="xsd:string" minOccurs="1"/>
        <xsd:element name="description" type="xsd:string"
             minOccurs="0"/>
        <xsd:element name="title" type="xsd:string" minOccurs="0"/>
        <xsd:element name="author" type="xsd:string"
             minOccurs="0" maxOccurs="unbounded" />
        <xsd:element name="version" type="xsd:string"/>
        <xsd:element name="previousVersion" type="xsd:anyURI"
             minOccurs="0" />
        <xsd:element name="lastModified" type="xsd:dateTime" />
        <xsd:element name="import" type="ModelProxy"
             minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="remotemodel" type="RemoteModel"
              minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="package" type="Package"
             minOccurs="0" maxOccurs="unbounded"/>
        <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:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

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

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

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

### 4.7.4 version : string [0..1]
Label indicating the version of this model.

### 4.7.5 previousVersion : anyURI [0..1]
URL to the VO-DML/XML document with the previous version of this data model from which the current version was derived.

### 4.7.6 import : ModelProxy[→] [0..*]
For a Model to reuse types form another model, or to indicate identifications between its elements and an external model, a ModelProxy must be created through an *import* element.

### 4.7.7 remoteModel : RemoteModel[→] [0..*]
A 'remoteModel' element indicates that information in the current model uses some remote model, but in a loose manner. For example a concept in the current model may be declared to be "similar" to a concept defined in a remote model, where this remote model need *not* be a VO-DML model, but may ba an XML schema for example.

### 4.7.8 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.

### 4.7.9 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”.

### 4.7.10 dataType : DataType[→] [0..*]
Collection of DataType-s defined directly under the model.

### 4.7.11 primitiveType : PrimitiveType[→] [0..*]
Collection of PrimitiveType-s defined directly under the model.

### 4.7.12 enumeration : Enumeration[→] [0..*]
Collection of Enumeration-s defined directly under the model.

## 4.8 Package extends ReferencableElement[→]

### 4.8.1 name: string [1]
The name of the package, which must be unique in the parent container of the package, either the model, or another package. The uniqueness is determined over all child elements of the container at the same level, both types and other packages.

### 4.8.2 description: string [0..1]
A short description of this package.

### 4.8.3 depends : PackageDependency[→] [0..*]
A package can "depend" on other packages. This generally implies that types in one package depend on one or more types in the other package.

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

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

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

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

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

## 4.9 ModelProxy extends *ReferencableElement* [→]
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. This "proxy" provides a URL identifying the VO-DML/XML representation of the imported model, as well as a URL to the documentation of that model. It also MAY define a prefix that is to be used when interpreting ElementRef elements in the current model, but see the discussion in the section on the utyperef element in ElementRef[→].

```
<xsd:complexType name="ModelProxy">
  <xsd:complexContent>
    <xsd:extension base="ReferencableElement">
      <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" minOccurs="1">
        <xsd:element name="prefix" type="xsd:string" minOccurs="0"/>
        <xsd:element name="documentationURL" type="xsd:anyURI"
             minOccurs="1"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

## 4.10 *Type* extends *ReferencableElement* [→]
This is the base class of all type definitions. A Type is a ... [TBD]

### 4.10.1 Name
The name of the type. Must be unique in the collection of names of types+packages in the direct container (Model or Package) of the Type.

### 4.10.2 Description
Free form description of the Type.

### 4.10.3 extends: TypeExtension [0..1]
If not null, defines the base type of this Type.

## 4.11 ObjectType extends *Type* [→]
Represents the ObjectType concept defined in section 3.2.2 .

### 4.11.1 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.]

### 4.11.2 attribute: Attribute[→] [0..*]

Collection of Attribute definitions.

### 4.11.3 reference: Reference[→] [0..*]

### 4.11.4 collection: Collection[→] [0..*]

## 4.12 *ValueType* extends *Type* [→]

## 4.13 DataType extends *ValueType* [→]

## 4.14 PrimitiveType extends *ValueType* [→]

PrimitiveTypes are the simplest examples of ValueTypes. They are represented by a single value only. A set of PrimitiveTypes is predefined in the IVOA_Profile data model in https://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/profile/IVOA_Profile.vo-dml.xml .

## 4.15 Enumeration extends *ValueType* [→]

## 4.16 Literal extends *ReferencableElement* [→]

## 4.17 *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>
```

### 4.17.1 datatype : ElementRef

The "datatype" element identifies the target element, which MUST be a Type.

## 4.18 Attribute extends *Role* [→]

**Rules**:

- The datatype of an Attribute MUST identify a ValueType

### 4.18.1 name : string [1]

The name of the attribute, which must be unique within the collection of named roles available on a type. This includes inherited roles.

### 4.18.2 description : string [1]

Description of the attribute.

### 4.18.3 multiplicity : Multiplicity[→] [1]

The multiplicity (also sometimes referred to as cardinality) indicates whether an attribute must have a value, or not, or whether it may have multiple values.

### 4.18.4 constraints : Constraints [0..1]

ddd

### 4.18.5 skosconcept : SKOSConcept [0..1]

If an Attribute defines a "skosconcept", it indicates that its values should represents 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 datatype attribute should be compatible with a string.

## 4.19 Container extends *Role*

Reference from an objecttype to another objecttype that to

## 4.20 TypeExtension extends *Role*

TypeExtension represents the "usual" inheritance relationship familiar from object oriented programming and modeling. In VO-DML TYpeExtension is a role, hence has a datype element that identifies the base type through its value.

For data models it implies that properties are inherited from base type

## 4.21 *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.

### 4.21.1 name : string [1]

### 4.21.2 description : string [1]

### 4.21.3 multiplicity : Multiplicity [1]

## 4.22 Reference extends *Relation*

## *4.23* Collection extends *Relation*

A collection represents a composition relationship between a parent and child ObjectType.

## 4.24 Identification extends *Relation*

This class indicates a semantic relation between a concept defined in the current model and the target concept of the relation. How to interpret the relation is indicated by the attributes of the identification object.
[TBC]

## 4.25 Multiplicity

Multiplicity indicates the cardinality of roles. It is implemented as an enumerated list with the following values:

- 1
  there must be 1 and only 1 value assigned.
- 0..1
  at most one value can be assigned, possibly none.
- 0..*
  any number of values can be assigned, including none.
- 1..*
  there must be at least 1 value assigned, possibly more.

# 5 Mapping to serialization formats

## 5.1 XSD

## 5.2 RDB

## 5.3 Java

## 5.4 VOTable

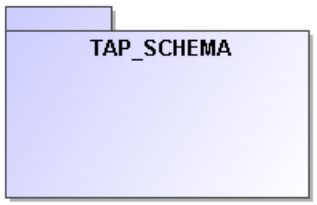
# Appendix A Graphical representation

When showing example data models we will often use a graphical representation of the model. For this we use a UML "syntax" with some extra rules applied. The VO-URP pipeline[9], and the xmi2vo-dml.xsl script work with a MagicDraw CommunityEdition 12.1 UML *Profile*. This was also used in the Simulation Data Model. The XSLT script will recognize the UML concepts from the profile and will map these to corresponding VO-DML concepts. Here we give for each VO-DML concept an example UML component:

## A.1 Model

Not explicitly represented.

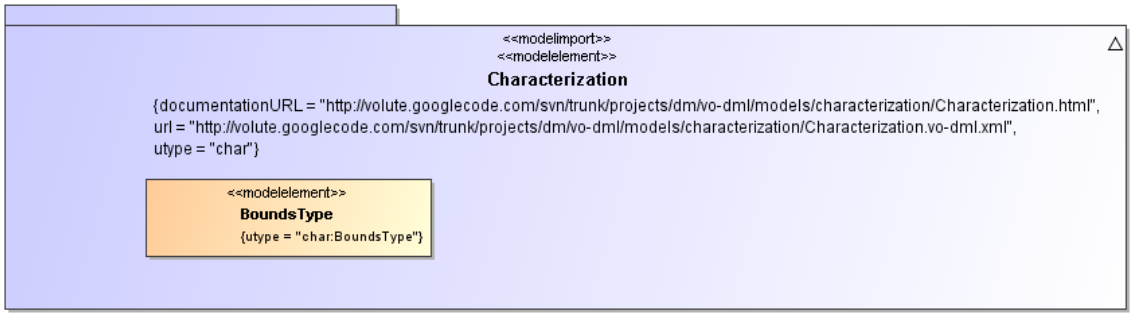
## A.2 Package




## A.3 ModelImport

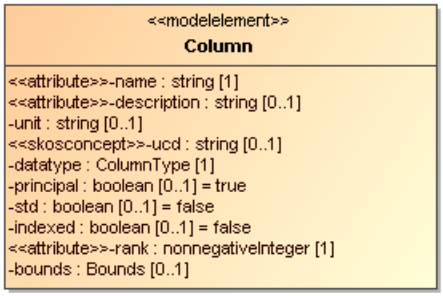
A package-like element with stereotype <<modelimport>> and possibly some contained type proxies. The latter MUST define a utype.

[9] See presentations in Triest 2008: http://wiki.ivoa.net/internal/IVOA/InterOpMay2008DataModels/dmstandards.ppt and http://wiki.ivoa.net/internal/IVOA/InterOpMay2008DataModels/IVOA-InterOp2008-UML-VO-Transformer-1_0.pdf and GoogleCode site, http://vo-urp.googlecode.com and




## A.4 ObjectType

A box which may have a stereotype such as <<modelelement>> in example, but may go without one. May have attributes.

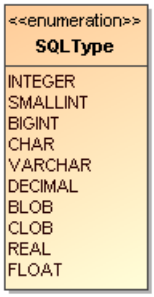



## A.5 PrimitiveType

A box with stereotype <<primitivetype>> and a name. Nothing else.

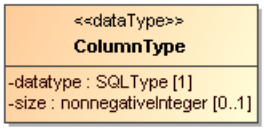
## A.6 Enumeration

A box with stereotype <<enumeration>> and a list of literals.




## A.7 DataType

A box with stereotype <<datatype>> and attributes.



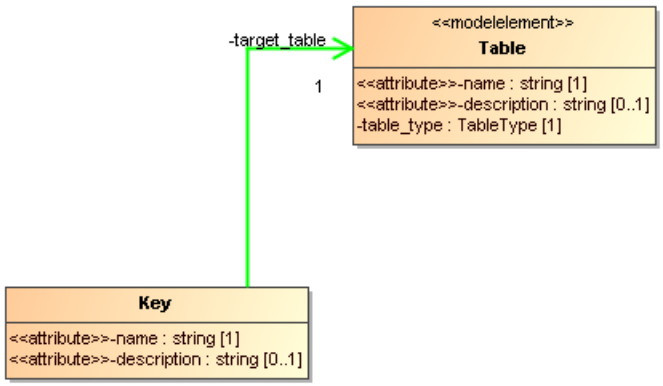

## A.8 Attribute

See DataType or ObjectType for examples. The entries in the bottom part of the box are attributes. They must have a name and data type separated by a ':' and

must have a multiplicity indicated by the expression between '[]'. They may have a stereotype.

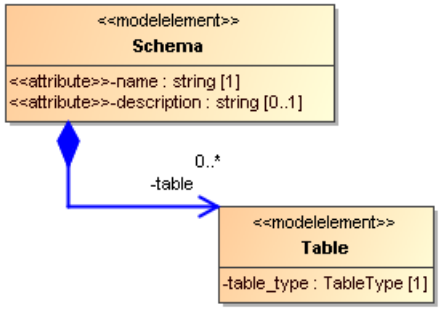
## A.9 Reference

A reference is represented by a (green) arrow from a source ObjectType or DataType to a target ObjectType. The end near the target has a name and multiplicity.



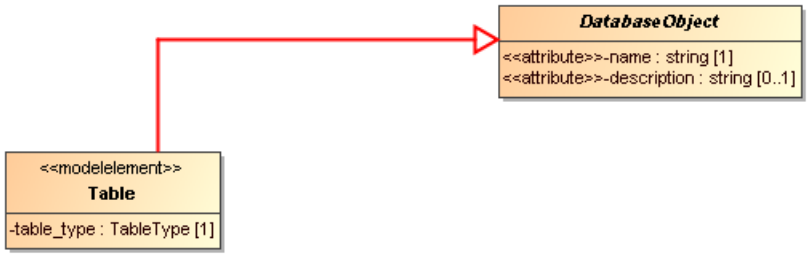

## A.10 Collection and Container

A collection is represented by a (blue) line with a diamond attached to a parent ObjectType and an arrow attached to the child ObjectType. The end near the child has the collection's name and its multiplicity.




The parent end is not given an explicit name, but is referred to as the Container for the child type.

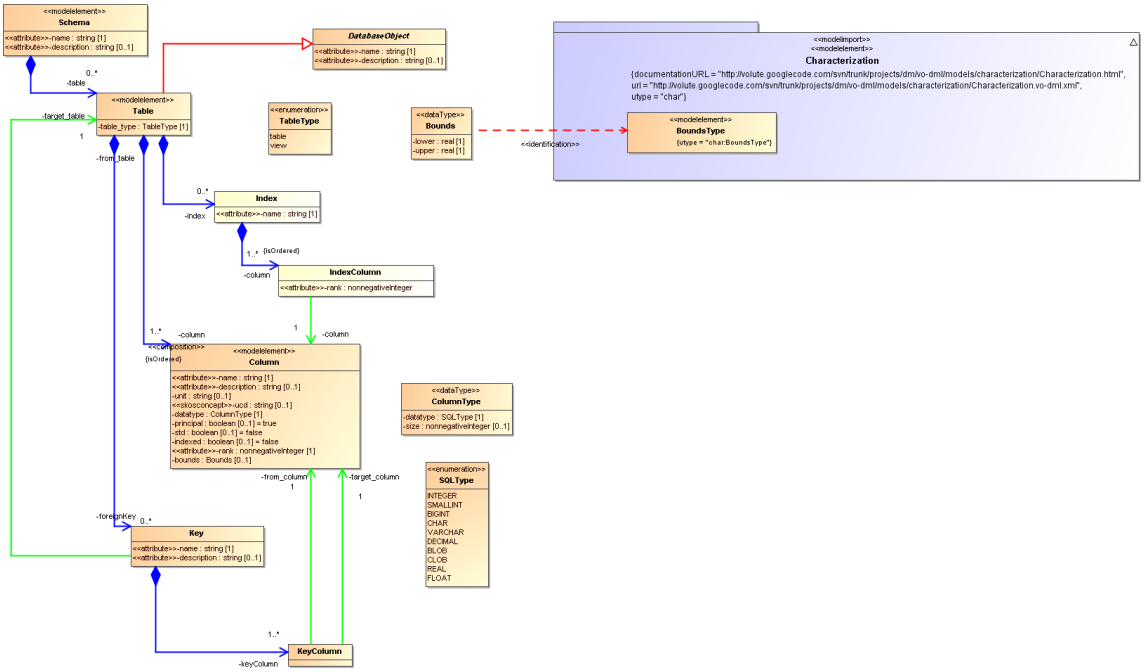
## A.11 TypeExtension



A red, open arrow from class Table to class DatabaseObject indicates that the former extends, or "inherits from" the latter, which is also the "base type" of the former.

## Appendix B TAP in VO-DML

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/tap .
The following figure shows a UML version of the model following the graphical rules explained in Appendix A.




## Appendix C How to do data modelling

Previous work

- http://www.ivoa.net/internal/IVOA/InterOpMay2004DataModel/dm-presentation20040528.ppt
- http://www.ivoa.net/internal/IVOA/VOResource010RevNotes/ModelBasedSchema.ppt
- http://wiki.ivoa.net/internal/IVOA/InterOpMay2008DataModels/dmstandards.ppt
-