International

Virtual

Observatory

Alliance

UTYPEs: Portable Data Model References

Version 1.0-20130509

Working Draft 2013 May 09

This version:
1.0-20130509
Latest version:
-
Previous version(s):

Editors:
Omar Laurino
Gerard Lemson

Authors:
UTYPEs tiger team: Patrick Dowler, Makus Demleitner, Matthew Graham, Omar Laurino, Gerard Lemson, Jesus Salgado.

Abstract

Data providers and curators provide a great deal of metadata with their data files: this metadata is invaluable for users and for Virtual Observatory software developers. In order to be interoperable, the metadata must refer to common Data Models. We propose a scheme for annotating data files in a standard, consistent, interoperable fashion, so that each piece of metadata can unambiguously refer to the correct Data Model element it expresses. We also describe in detail how to represent Data Model instances in the VOTable format. The mapping is operated through opaque, portable strings: UTYPEs.

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/.

Contents

1 Introduction 4
2 Use Cases 6
3 The need for a mapping language 7
4 The @utype attribute: Mapping types, roles, and inheritance. 9
4.1 The UTYPE format (Alternative 1) 10
4.2 The UTYPE format (Alternative 2) 11
4.2.1 The Namespace URI 11
4.2.2 How to look for a UTYPE in a document 11
4.2.3 Extended notation for portable use: UTYPEs as URIs 12
4.3 The VO-DML preamble 12
5 Examples: Mapping VO-DML ⇒ VOTable 12
5.1 Sample model and instances 13
5.2 Data carriers in VOTable 16
5.3 Mapping ObjectType 16
5.4 Mapping Attribute 18
5.5 Mapping Reference 20
5.6 Mapping Collection 21
5.7 Mapping value types 24
5.8 Mapping Inheritance 25
6 Patterns for annotating VOTable: specification 27
6.1 Model 28
6.1.1 Model to GROUP in VOTABLE 28
6.2 DataType 29
6.2.1 DataType to GROUP in RESOURCE 30
6.2.2 DataType to GROUP in TABLE 31
6.2.3 DataType to GROUP in a GROUP with no @utype 31
6.3 Attribute 32
6.3.1 Attribute to FIELDref in GROUP 32
6.3.2 Attribute to PARAM in GROUP 32
6.3.3 Attribute to PARAMref in GROUP 33
6.3.4 Attribute to GROUP in GROUP 33
6.4 ObjectType 34
6.5 Reference 34
6.5.1 Reference (from Object|DataType to ObjectType) to GROUP 34
6.5.2 Reference (from Object|DataType to ObjectType) to TR 35
6.6 Collection 35
6.6.1 Collection.item to GROUP in GROUP 36
6.7 Extends, inheritance 36
6.8 Value, Unit, UCD 37
6.9 ORM Mapping Patterns [TBD] 37
7 Notable absences 38
7.1 Atomic Types: support for custom and legacy UTYPEs 38
7.2 Packages 39
7.3 ObjectType to TABLE 39
7.4 Attribute to FIELD|PARAM in TABLE 39
8 Serializing to other file formats [TBD] 39
Appendix A. List of all valid mapping patterns [TBD] 40
Appendix B. Growing complexity: naïve, advanced, and guru clients 40
Appendix C. Frequently Asked Questions 41

1 Introduction

Data providers put a lot of effort in organizing and maintaining metadata that precisely describes their data files. This information is invaluable for users and for software developers that provide users with user-friendly VO-enabled applications. For example, such metadata can characterize the different axes of the reference system in which the data is expressed, or the history of a measurement, like the publication where the measurement was drawn from, the calibration type, and so forth. In order to be interoperable, this metadata must refer to some Data Model that is known to all parties: the IVOA defines and maintains such standardized Data Models that describe astronomical data in an abstract, interoperable way.

We will argue that, in order to enable such interoperable, extensible, portable annotation of data files, one needs:

i) Pointers linking a specific piece of information (data or metadata) to the Data Model element it represents (UTYPEs).

ii) A language to describe Data Models and their valid pointers, so to support extensibility and efficient software development (VO-DML)

iii) A mapping specification that unambiguously describes the mapping patterns

Without a consistent language for describing Data Models, pointers alone are ambiguous and redundant: as such, they may have limited value. On the other hand, the language must be expressive and formal enough to enable the serialization of data types of growing complexity and the development of reusable, extensible software components and libraries that can make the technological uptake of the VO standards seamless and scalable.

Also, one needs to map the abstract Data Model to a particular format meta-model. For instance, the VOTable format defines RESOURCEs, TABLEs, PARAMs, FIELDs, and so forth, and provides explicit attributes such as units, UCDs, and utypes: in order to represent instances of a Data Model, one needs to define an unambiguous mapping between these meta-model elements and the Data Model language, so to make it possible for software to be able to parse a file according to its Data Model and to Data Providers to mark up their data products.

While one might argue that a standard for portable, interoperable Data Model representation would have been required before one could think about such a mapping, we are specifying it only at a later stage. This means that several different interpretations of UTYPEs have been proposed and used: some explicitly require the parsing of the UTYPEs strings, others make UTYPEs very redundant, by defining many UTYPEs (as many as hundreds) for expressing the same concept (e.g. the accuracy of a measurement) in different contexts. For instance, the accuracy of the measurement on the spectral axis, the accuracy of the measurement on the time axis, the overall accuracy on the spectral axis for a dataset, an accuracy provided by a service response or the one provided by a standalone file, all have different UTYPEs, while an error bar is

an error bar regardless of the quantity to which it is referred, or whence the response originally came from[1].

Any standard trying to reconcile these very different usages must take them into account and make the transition from the current usages to the new standard as seamless as possible. For this reason, this document also shows how the current UTYPEs usages can be seamlessly integrated with the new scheme, so to minimize the transition effort.

Actually, since this standardization is required by new, more complex Data Models (such as data cubes and their projections, like SEDs or Time Series) than the ones already defined, it will be possible to adopt this standard only in the new Data Models and Data Access Services, thus avoiding any transition efforts. Legacy services can *add* metadata to their files in order to make them compatible with the new standard, but they do not need to *change* them in such a way that would make them incompatible with existing software.

This is a very technical and formal document that can enable the development of flexible, reusable, user-friendly libraries and applications that abstract and generalize the input/output access to VO compliant files, thus facilitating the uptake of the VO in the astronomical community, and the simplification of the process for publishing data to the VO. However, several sections of this document are utterly informative: in particular, the appendices provide more information about the impact of this specification to the current and future IVOA practices.

This document defines the scope and usage of standardized UTYPEs as means to annotate and describe Data Model instances.

It also describes how to represent Data Model instances using the VOTable schema. This representation uses UTYPE attributes and the structure of the VOTable meta-model elements to indicate how instances of data models are stored in VOTable documents. We show many examples and give a complete listing of allowed mapping patterns. We believe this approach to be a complete and in a certain sense most explicit mapping language.

In sections 1-5 we give an introduction to the UTYPEs approach and several examples that illustrate the mapping.
Section 6 aims to be a more rigorous listing of all valid annotations. Appendix A contains the complete list of the mapping patterns supported by this specification. Section 7 describes what patterns and usages this specification doesn't cover; moreover, it describes how legacy and custom UTYPEs can be treated in this specification's framework: as such, this section actually describes the *transition* from the current usages and this specification.

To illustrate both the problem and the solution, throughout the document we will refer to some real or exemplar Data Models. Please remember that the example Data Model have been designed to be fairly simple, yet complex enough to illustrate all the possible constructs that this specification covers. They are not to

[1] See the Current Usages Note:
http://www.ivoa.net/documents/Notes/UTypesUsage/20130213/NOTE-utypes-usage-1.0-20130213.pdf

be intended as actual DMs, nor, by any means, this specification suggests their adoption by the IVOA or by users and or Data Providers. In some cases we refer to actual DMs in order to provide an idea of how this specification relates to real life cases involving actual DMs.

2 Use Cases

The use cases enabled by this mapping definition are limitless. This bold statement can be easily validated by considering that what we describe is analogous to the natural mapping between Data Models and XSD schemata, where instances are expressed in XML documents. XML is widely used in so many ways that it is impossible to list them all. As a matter of fact, XML can even express lists of its own use cases.

However, to give a sense of what it is possible to accomplish with this specification, we provide some explicit use cases relative to the VO domain.

Find a value by UTYPE. Given a VOTable annotated using UTYPEs, a client can extract a piece of information by finding a PARAM or FIELDref annotated with a predefined UTYPE. For example, the client can find the luminosity measurement by looking for the element with UTYPE src:source.LuminosityMeasurement.value. The predefined UTYPE is invariant in the Data Models that *reuse* a particular concept.

Find an object by UTYPE(s). Given a VOTable annotated using UTYPEs, a client can build in memory the instance of an Object defined by some Data Model, assuming the knowledge of a finite set of UTYPEs. For example, the client can find all the information about a Source by looking at a GROUP annotated with the UTYPE src:source.Source, and interpret its components (PARAMs and FIELDrefs) as the attributes of the object, identified by their UTYPE strings.

Translation from VOTable to Data Model instance. A universal translator interpreter can parse Data Model descriptions and a VOTable document and translate the document in a valid set of instances, by using UTYPEs and a standard specification for model-to-model mapping.

VO-enabled plotting and fitting applications. An application whose main requirement is to display, plot, and/or fit data cannot be required to be aware of *all* data models. However, if these data models share some common representation of quantities, their errors, and their units, the application can discover these pieces of information and structure a plot, or perform a fit, with minimal user input: each point will be associated with an error bar, upper/lower limits, and other metadata. The application remains mostly Data Model-agnostic: it wouldn't need to *understand* concepts like Spectrum, or Photometry.

Validators. The existence of an explicit Data Model representation language and of a precise, unambiguous mapping specification using UTYPEs enables the creation of universal validators, just as it happens for XML and XSD: the validator can parse the Data Model descriptions imported by the VOTable and check that the file represents valid instances of the Data Model.

VO Publishing Helper. A universal publisher application allows Data Providers to interactively create custom Data Models importing the standard IVOA ones, and to represent them in a standardized description language. The application also helps Data Providers in interactively mapping Data Models elements to their files or DB tables, either producing a VOTable template with the appropriate UTYPEs annotation, or by creating a DAL service on the fly. The VO Publisher application is not required to be DM-aware, since it can get all the information from the standardized description files.

VO Importer. Users and Data Providers have files that they want to make VO-compliant: a DM-unaware Importer application allows them to convert the file to a supported format. After that, or if the file is already in a supported format, it allows the user to interactively map the data and metadata in their files to a standardized DM representation, using UTYPEs for annotating the file.

Extensibility. Most often each astronomical facility, instrument, or mission needs to express measurements and metadata attributes that are unique to the facility, instrument, or mission. A Data Provider may want to *extend* a Data Model, adding to the common information about astronomical sources and data products the metadata that is specific for their instruments or domain. The added metadata can be described in a standardized fashion so that the user can take advantage of the information.

3 The need for a mapping language

When encountering a data container, i.e. a file or database containing data, one may wish to interpret its contents according to some external, predefined data model. That is, one may want to try to identify and extract instances of the data model from amongst the information. For example in the "global as view" approach to information integration, one identifies elements (e.g. tables) defined in a global schema with views defined on the distributed databases[2].

If one is told that the data container is structured according to some standard serialization format of the data model, one is done. I.e. if the local database is an exact *implementation* of the global schema, one needs no special annotation mechanism to identify these instances. An example of this is an XML document conforming to an XML schema that is an exact physical *representation* of the data model.

But in an information integration project like the IVOA, which aims to homogenize access to many distributed heterogeneous data sets, databases and documents are in general *not* structured according to a standard representation of some predefined, global data model. The best one may hope for is to obtain an *interpretation* of the data set, defining it as a *custom serialization* of the result of a *transformation* of the global data model[3]. For example, even if databases themselves are exact replications of a global data model, results of general queries will be such custom serializations.

To interpret such a custom serialization one generally needs extra information that can provide a *mapping* of the serialization to the original model. If the serialization *format* is known, this mapping may be given in phrases containing elements both from the serialization format and the data model. For example if our serialization contains data stored in 'rows' in one or more 'tables' that each have a unique 'name' and contain 'columns' also with a 'name', you might be able to say things like:

[2] See, for example, http://logic.stanford.edu/dataintegration/chapters/chap01.html

[3] Or alternatively as a transformation of a (standard) serialization of the data model.

- The rows in this table named SOURCE contain instances of object type 'Source' as defined in data model 'SourceDM' **(SourceDM is an example model formally defined later in this document)**.
- The type's 'name' attribute (having datatype 'string', a primitive type) also acts as the identifier of the Source instances and is stored in the single column with name ID.
- The type's 'classification' attribute is stored in the table column CLASSIFICATION (from the data model we know its datatype is an enumeration with certain values, e.g. 'star', 'galaxy', 'agn').
- The type's 'position' attribute (being of structured data type 'SkyCoordinate' defined in model 'SourceDM') is stored over the two columns RA and DEC, where RA stores the SkyCoordinate's attribute 'longitude', DEC stores the 'latitude' attribute. Both must be interpreted using an instance of the SkyCoordinateSystem type, This instance is stored in 1) another document elsewhere, referenced by a reference to a URI, or 2) in this document, by means of an identifier.
- Instances from the collection of luminosities of the Source instances are stored in the same row as the source itself. Columns MAG_U and ERR_U give the 'magnitude' and 'error' attributes of type LuminosityMeasurement in the "u band", an instance of the Filter type. (stored elsewhere in this document ('a reference to this Filter instance is ...'). Columns MAG_G and ERR_G ... etc.
- Luminosity instances also have a filter relation that points to instances of the PhotometryFilter structured data type, defined in the IVOA PhotDM model, whose package is imported by the SourceDM.

In this example the underlined words refer to concepts defined in VO-DML, a meta-model that is used as a formal language for expressing data models. The use of such a modeling language lies in the fact that it provides formal, simple and implementation neutral definitions of the possible structure, the 'type' and 'role' of the elements from the actual data models that one may encounter in the serialization (SourceDM). This can be used to constrain or validate the serialization, but more importantly it allows us to formulate mapping rules between the serialization format (itself a kind of meta-model) and the meta-model, independent of the particular data models used; for example rules like:

- An object type MUST be stored in a 'group'.
- A 'primitive type' MUST be stored in a 'column'.
- A reference MUST identify an object type instance represented elsewhere, either in another 'table', possibly in the same table, possibly in another document.
- An attribute SHOULD be stored in the same table as its containing object type.
- etc

Clearly free-form English sentences as the ones in the example are not what we're after. If we want to be able to identify how a data model is represented in some custom serialization we need a formal, computer readable mapping language.
One part of the mapping language should be anchored in a formally defined serialization language. After all, for some tool to interpret a serialization, it MUST understand its format. A completely freeform serialization is not under consideration here. This

document assumes VOTable, even though a discussion on other formats is provided in Section 8.

The mapping language must support the interpretation of elements from the serialization language in terms of elements from the data model. If we want to define a generic mapping mechanism, one by which we can describe how a general data model is serialized inside a VOTable, it is necessary to use a general data model *language* as the target for the mapping, such as the one described above. This language can give formal and more explicit meaning to data modeling concepts, possibly independent of specific languages representation languages such as XML schema, Java or the relational model. This document uses VO-DML as the target language.[4]

The final ingredient in the mapping language is a mechanism that ties the components from the two different meta-models together into "sentences". This generally requires some kind of explicit annotation, some meta-data elements that provide an identification of source to target structure. The 'utype' VOTable attribute can provide this link in a rather simple manner:

- The value of a utype attribute must correspond to the VODML-ID identifier of an element explicitly defined in VO-DML/XML.
- The VOTable element owning the utype attribute is said to *represent* the identified VO-DML data model element. It identifies one or more instances of the data model element, the identification depends on the kind of element and on the context in which it appears.
- There is a set of rules that constrain *which* VOTable elements can be identified with *which* type of VO-DML element and how the context plays a role here.

This solution is sufficient and it is in some sense the simplest and most explicit approach for annotating a VOTable. It may *not* be the most natural or suitable approach for other meta-models such as FITS or TAP_SCHEMA. For example the current approach relies heavily using on GROUPs to identify most of the structural mapping. FITS and TAP_SCHEMA do not currently possess such a construct. We will discuss this at the end of this document.

4 The @utype attribute: Mapping types, roles, and inheritance.

A mapping pattern from VOTable to VO-DML uses a @utype attribute of a VOTable element to identify the role of the VO-DML element it represents. These @utype-s always point to an *role* definitions.

A single attribute in a Data Model is characterized by its *role* in the Data Model itself and by its *type*. The role of the attribute allows a reader to attach the instance as an attribute of the right element. Its type allows the reader to *cast* the instance to the right class. The

[4] A complete reference of VO-DML is provided by this document (please note that the document is still in a state of flux): http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/doc/VO-DML-WD-v.0.x.pdf

importance of these annotations may vary according to the programming language and/or use case, but including complete and explicit annotations for both the type and role of each property enables the maximum flexibility.

A typical usage scenario may be a VOTable naïve (see Appendix B) client that is sensitive to certain models only, say STC. Such a tool can be written to understand annotation with STC types. Finding an element mapped to a type definition from STC it might infer for example that it represents a coordinate on the sky and use this information according to its requirements.

Such a tool would not necessarily understand other models where such an STC type is *used* as a role. So, if the annotation refers to both the attribute's role *and* type, even a naïve client can trivially find the information it needs. A more advanced client may want to read the Data Model Description File that describes the Data Model in a standardized, machine readable, fashion.

Other scenarios involve inheritance and polymorphism. Inheritance allows models to extend classes defined in other data models. Polymorphism is the common object-oriented design concept that says that the declared type of a property may not be the same as the type of an instance of that property that is actually serialized. In particular, the value of a property may be an instance of a *subtype* of the declared type. So in general it is not enough to know the type of the attribute (for example) to uniquely know which type of instance to expect. And it may also not be possible to infer the instance type uniquely from the contents of the element representing the attribute.

Hence a single @utype attribute with value indicating only the role may not be sufficient to infer all DM information about a VOTable element.[5]

Typed languages such as Java support a casting operation, which provides more information to the interpreter about the type it may expect a certain instance to be.

The following two sections describe two different scenarios for the UTYPEs format: they both have pros and cons, and we need more discussion and feedback from the community in order to adopt only one of the two.

4.1 The UTYPE format (Alternative 1)

Prefixes are sequences of [A-Za-z0-9_-], and they are case sensitive.

It is recommended to form non-REC DM prefixes as <author-acronym>_<dm-name>; thus, NED's derivation of spec could have ned_spec as a prefix, CDS's derivation cds_spec.

[5] In fact, we deem utype should actually be a VOTable element with a structure of its own, rather than an attribute, in order to effectively represent the complex information it has to carry. However, we decided to find a solution that would not require any changes to other specifications. An alternative solution might have been to use utypes concatenation, but this was deemed as too complicated for naïve clients, even though such mechanism is already in place for UCDs. The solution employed here is the best solution we could come up with, given the above limitations. It is less elegant and compact, but it conveys all the needed information.

Prefixes correspond to major versions for the corresponding data models. Thus, utypes remain constant over "compatible" changes in the sense of [DOCSTD]. In consequence, clients must assume a compatible extension when encountering an unknown utype with a known prefix (and should in general not fail).

Another consequence of this rule is that there may be several VO-DML URLs for a given prefix. To identify a data model, use the prefix, not the VO-DML URL, which is intended for retrieval of the data model definition exclusively. In case a client requires the exact minor version of the data model, it must inspect SOMETHING ELSE.

4.2 The UTYPE format (Alternative 2)

The VOTable specification defines the @utype attribute and its requirements. Documents implementing this specification, however, MUST comply with a restricted syntax. When we use the terms UTYPEs and @utype we will refer to string complying with this specialized syntax.

UTYPEs have the form:

```
UTYPE ::= prefix ':' localname
```

Prefixes are sequences of [A-Za-z0-9_-], and they are case sensitive.

Localnames are also case sensitive and they have the same syntax of URI fragments.

UTYPEs are always considered opaque, meaning that clients have no reason to parse them. They are identifiers mapping VOTable elements to VO-DML elements through their VODML-ID. Thus, they must follow the same syntax rules defined in the VO-DML/Schema document.

4.2.1 The Namespace URI

The UTYPE prefix is a reference to a namespace URI defined in a VO-DML preamble (see 4.3).

The namespace URI MUST be an IVOA Resource Name (IVORN) in the form ivo://authorityID/DM-ID

4.2.2 How to look for a UTYPE in a document

Clients may match UTYPEs by the simple string comparison of the @utype attribute value in a VOTable with UTYPEs defined in the Data Models descriptions.

Clients need to look for the Data Models they are interested in by parsing the VO-DML preamble (see 4.3) and matching the Model's URI with the ones declared in the preamble. The preamble maps the Model to a prefix string. This string must be attached to the id part according to the prefix:localname syntax before it can be compared to the UTYPEs in the document.

4.2.3 Extended notation for portable use: UTYPEs as URIs

In order to make UTYPEs portable outside of the VOTable @utype semantics, we define an extended URI notation for UTYPEs by stringing together the namespace URI and the local name of their QName compact notation: the local name will be the fragment part of the URI.

Thus, the extended notation for a UTYPE with local name 'local' will be:
ivo://authorityID/DM-ID#local

Such IDs can be referenced in any context and can be resolved to the VO-DML document description using the standard mechanism for resolving IVORNs in Resource Registries.

4.3 The VO-DML preamble

In order to signal the reader that a document falls under this specification, a VOTable instance MUST declare all the Data Models it includes, the UTYPEs prefixes for this model, and the URI of the Data Model. They SHOULD also declare the actual URL of the VO-DML/XML description as a shortcut.

Any number of such declarations can be included in a document.

```
<GROUP utype="vo-dml:Model" name="src">
   <PARAM     utype="vo-dml:Model.uri"     name="uri"     datatype="char"     arraysize="*"
value="ivo://ivoa.net/SourceDM" />
   <PARAM     utype="vo-dml:Model.url"     name="url"     datatype="char"     arraysize="*"
value="https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/sample/Source.vo-dml.xml" />
</GROUP>
```

The above example introduces the Source Data Model, assigns the prefix "src" (the Model name) to its UTYPEs, and refers to the VO-DML/XML description of the Data Model. Notice the "vo-dml:Model" special utype that annotates the GROUP element to introduce the declaration. Notice that the vo-dml: UTYPEs point to an implicitly declared actual VO-DML/XML description.

The preamble GROUPs MUST be direct children of the VOTABLE element.

The following section shows examples of mappings as they may occur in realistic VOTables, using example Data Models. We will describe how the different VO-DML elements must be serialized and how annotations employing the @utype attribute and other meta-data elements can help one interpret the VOTable. In the later normative section we turn this around and explicitly list all legal annotations, their constraints and interpretation.

5 Examples: Mapping VO-DML ⇒ VOTable

In this section we list some mappings from VO-DML to VOTable. We use examples extracted from a sample model with sample instances described in the next section. These should be seen as an introduction to the complete and formal specification in

section 5.8 on how one SHOULD use utypes in VOTable to indicate mapping to VO-DML.

5.1 Sample model and instances

For examples we use a highly simplified version of a possible Source data model, illustrated by its UML representation in Figure 1. It has a VO-DML representation, which is reproduced in.

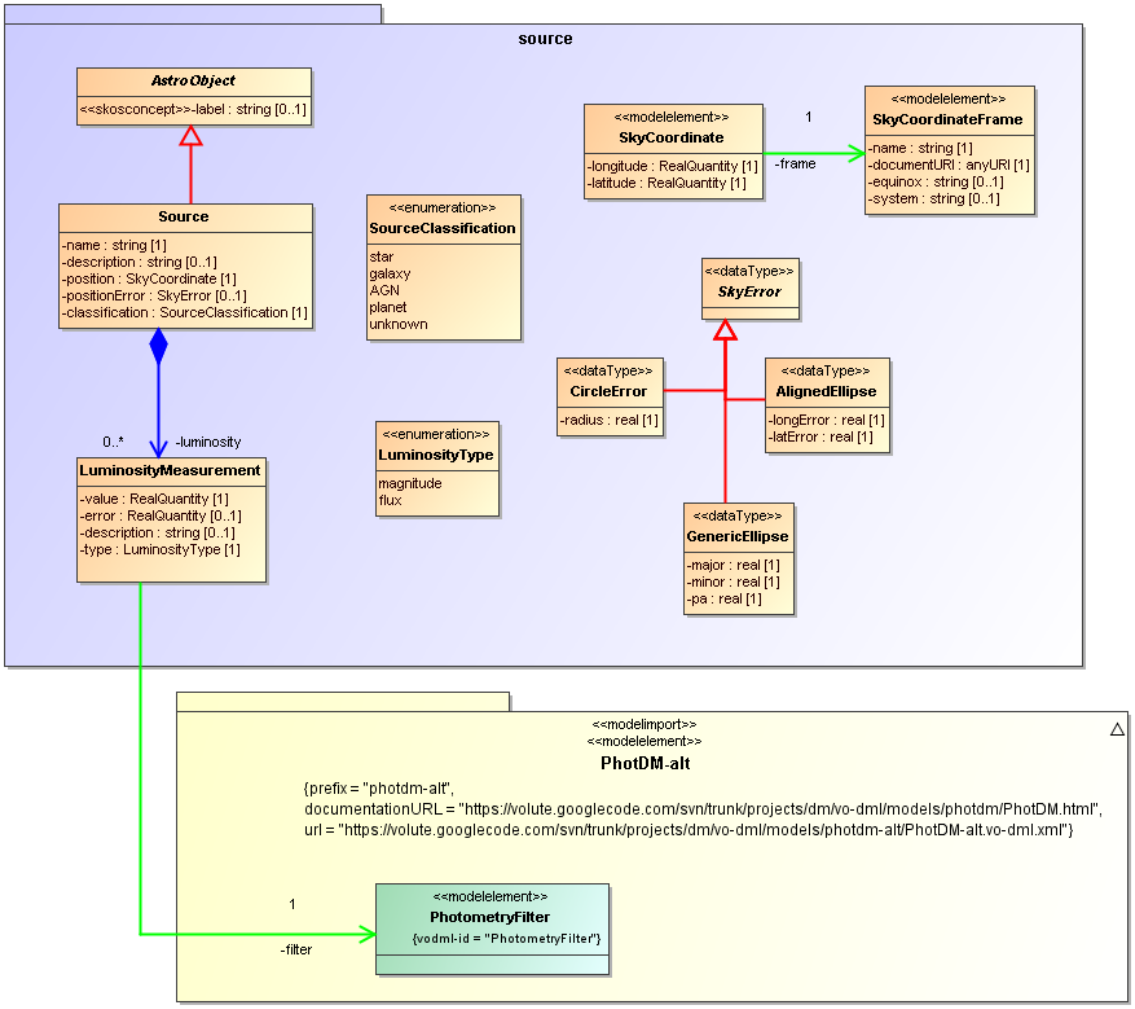


Figure 1 Example data model used in example. It represents a simplified Source data model, containing luminosities that refer to the imported PhotDM. It also defines a simplistic version of an STC model with some types for defining coordinates on the sky, for the sake of simplicity and just for example purposes.

The model defines some types allowing one to define a Source with position on the sky and a collection of luminosities. The position is modeled as a DataType, 'SkyCoordinate'. SkyCoordinate has a reference to a coordinate frame that is required to interpret its longitude and latitude attributes. The luminosities are really *measurements* of luminosities in a given filter that is indicated by a reference to a PhotometryFilter, which is imported from the PhotometryDM; hence they have a value *and* an error. A Quantity DataType is introduced that provides a real value and a unit.
The models are *by no means* meant to be comprehensive and include some admittedly artificial elements such as an Equinox PrimitiveType, which is supposed to be a simple

string and might carry enough semantic value of its own to use it as an annotation on PARAM elements for example.
Note that this sample model defines a Package that contains all the types. This package shows up in the values of the utype-s we use to identify the different elements. The values we use for these utype identifiers are generated from the VO-DML using a particular grammar: they are path-like expressions that are guaranteed to be unique and give some impression of the location in the data model.

We also use some sample instances of the models. These are here illustrated by UML instance diagrams. The diagram in Figure 2 represents the first two lines returned from a query to the SDSS DR7 database.

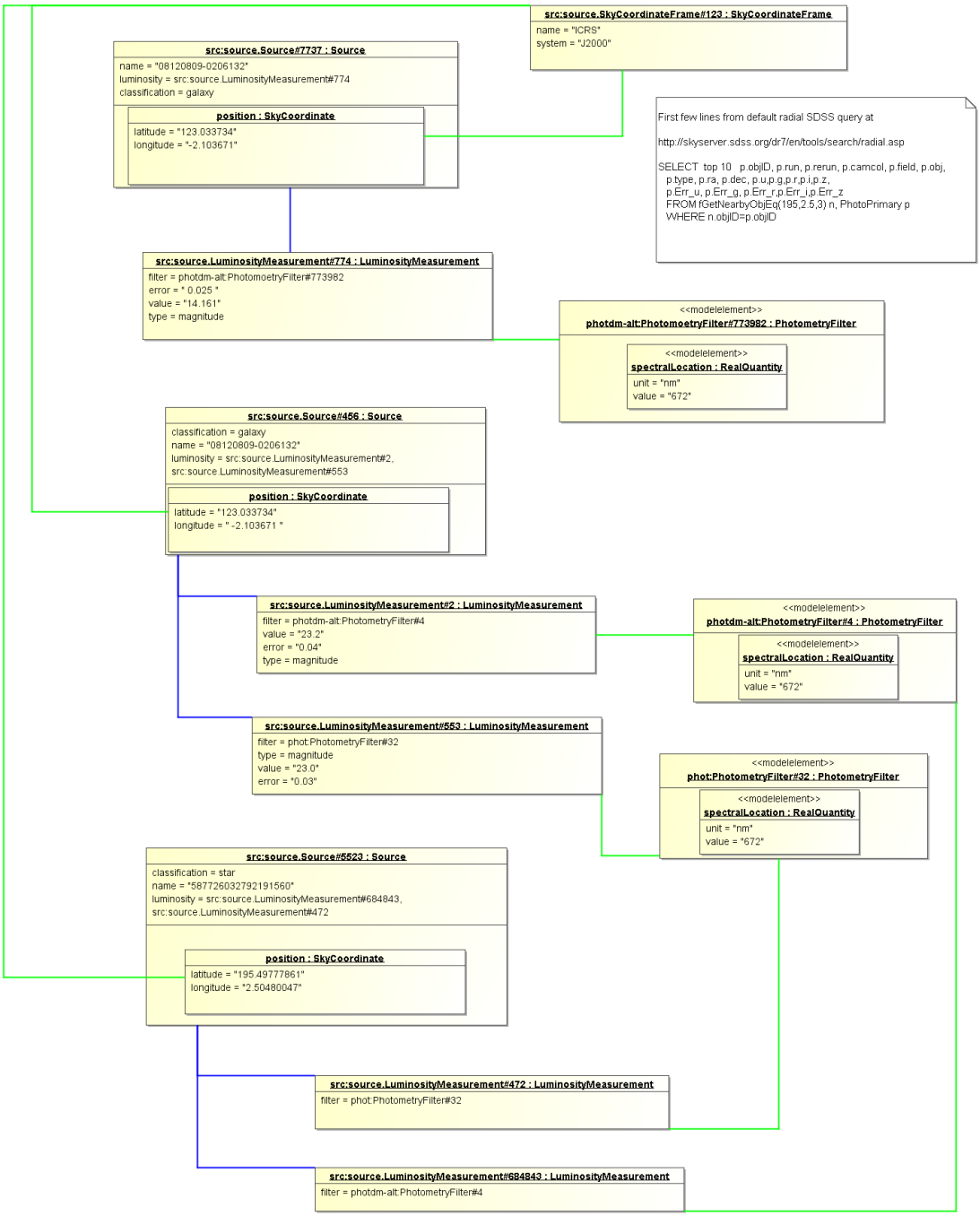


Figure 2 Instance diagram representing SDSS objetcs as sources in the sample data model. The first few results are represented from the default radial SDSS query at http://skyserver.sdss.org/dr7/en/tools/search/radial.asp corresponding to the SQL query:

```
SELECT  top 10   p.objID, p.run, p.rerun, p.camcol, p.field, p.obj,
   p.type, p.ra, p.dec, p.u,p.g,p.r,p.i,p.z,
   p.Err_u, p.Err_g, p.Err_r,p.Err_i,p.Err_z
   FROM fGetNearbyObjEq(195,2.5,3) n, PhotoPrimary p
   WHERE n.objID=p.objID
```

5.2 Data carriers in VOTable

VO-DML describes four different kinds of types: PrimitiveType (PT), Enumeration (E), DataType (DT) and ObjectType (OT). PT, E and DT are value types; OT is an object - or reference type. PT and E are atomic, their values consist of a single value; DT and OT are structured, they are built from multiple values, organized as attributes, and possibly of reference relations to OTs. An OT can also have collections of other OTs, and can have an identifier, an attribute of undetermined type that is implicitly defined for ObjectTypes.

To store instances (*values* and *objects*) of these types in a VOTable various options are available. Atomic values (i.e. instances of PT and E) are stored in cells in a row in a table (i.e. a TD) or in the value attribute of a PARAM (@value). To store an instance of a structured type one must store its components. To identify the structured instance in a serialization one must be able to identify the individual components and how they are to be combined. This last identification is done by the GROUP element. It is the main representation of structured types, both ObjectType and DataType. It is also used to represent relations to object types. These may be stored using foreign-key-like mechanisms or through some kind of hierarchy.

In fact in the approach described here virtually *all* mapping of VOTable to VO-DML is performed by GROUP elements and their components. They identify which elements are to be combined to create the object or DataType instance, what the roles are that these components play (attribute, reference), by simply using the utype attribute.

In the next few subsections we provide some examples how this mapping can be performed. It starts with the ObjectType and then discusses its components, Attribute, Reference and Collection. The mapping of the value types is discussed in the mapping of Attributes. The mapping of Reference and Collection relations is the most complex part of the whole mapping story and treated separately.

5.3 Mapping ObjectType

ObjectTypes consist of Attributes, References and Collections. How these are mapped is described in more detail below. But the important part for representing structured instance like an object is that these components must be combined together to construct a complete instance. In VOTable this is done using a GROUP element.

In the representation of ObjectType instances, GROUPs can be used in two different modes that are distinguished by the way the instance's data are ultimately stored. If all values are eventually stored exclusively in @value attributes of PARAM elements in the GROUP (or possibly outside the GROUP but accessed through PARAMrefs), the GROUP represents a complete instance *directly*. If even only one of the attributes is stored in a FIELD and accessed through a FIELDref, the GROUP is said to *indirectly* represent possibly multiple instances, one for each TR.

We will use these terms, "direct GROUP" and "indirect GROUP" as shorthand phrases for these representations all through the document. This freedom of choice *where* to store objects complicates the mapping of *relations* between objects, as many different referencing mechanisms must be taken into account. This is particularly important when discussing how to represent References in section 6.5

.

We illustrate the two modes of mapping by showing an example how each mode may represent exactly the same object. For this we use the object type in Figure 3 and a corresponding instance in Figure 4.

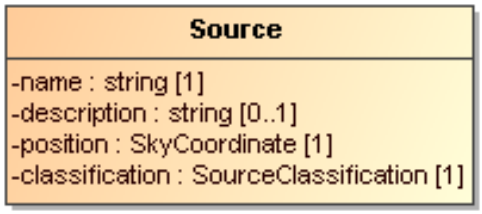


Figure 3 ObjectType representing a Source. [TBD Update]

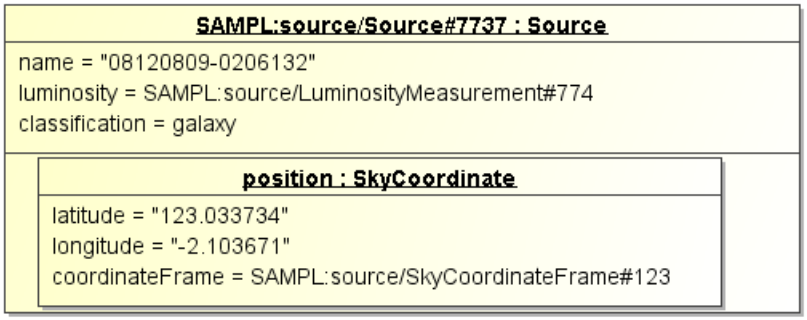


Figure 4 Instance of Source ObjectType. [TBD Update]

Indirect serialization to a TABLE:
Source instance from Figure 4 is stored in the first TR below. The TABLE is annotated using a GROUP with utype attribute set to “Instance.root”, which is a special role for root elements. The actual type is stored in the value of a PARAM with an Instance.type utype. Some atomic attributes are stored in FIELDs annotated by FIELDref-s, some in PARAMs; the child GROUP with its attributes annotated by FIELDref represents a structured attribute. Also, the SkyCoordinate.frame property is an example of GROUP reference. (for discussion see section 5.5):

```
<TABLE>
<GROUP utype="vo-dml:Instance.root" ID="_source">
      <PARAM     name="type"     utype="vo-dml:Instance.type"     value="src:source.Source"
datatype="char" arraysize="*"/>
      <FIELDref ref="_designation" utype="vo-dml:ObjectType.ID" />
      <FIELDref ref="_designation" utype="src:source.Source.name" />
      <GROUP utype="src:source.Source.position">
            <PARAM      utype="vo-dml:Instance.type"      value="src:source.SkyCoordinate"
name="datatype" datatype="char" arraysize="*"/>
            <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude" />
            <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude" />
            <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame" />
      </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD></TR>
...
</TABLE>
```

Note the special representation of the identifier of the Source object. This attribute is not defined explicitly in the model; hence no utype exists for it there. We can interpret each

ObjectType as ultimately being a subclass of some ObjectType class (just as in Java all classes ultimately extend java.lang.Object, generally implicitly). In VO-DML we can represent this by allowing an ObjectType to have a component of as yet unspecified type, which represents the ID attribute (implicitly) defined on this base class. We use a special utype for this attribute: vo-dml:ObjectType.ID, obtained from the VO-DML model discussed in the VO-DML reference document.

Direct serialization to a GROUP:
Here the instance is directly represented by a GROUP containing only PARAMs for the atomic attributes, and a GROUP with PARAMs for the structured attribute (and again a reference, see section 5.5).

```
<GROUP utype="vo-dml:Instance.root" >
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="src:source.Source"
datatype="char" arraysize="*"/>
  <PARAM utype="vo-dml:ObjectType.ID" value="08120809-0206132" .../>
  <PARAM utype="src:source.Source.name" value="08120809-0206132" .../>
  <PARAM utype="src:source.Source.classifiication" value="galaxy" .../>
  <GROUP utype="src:source.Source.position">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <PARAM utype="src:source.SkyCoordinate.longitude" value="123.033734" .../>
    <PARAM utype="src:source.SkyCoordinate.latitude" value="-2.103671" .../>
    <GROUP ref="_icrs" utype=" src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
```

Details on the mapping of the components are discussed next.

5.4 Mapping Attribute

An attribute is the role a value type plays in the definition of a structured type. They may represent atomic or may represent structured (Data)types themselves. These are represented differently.

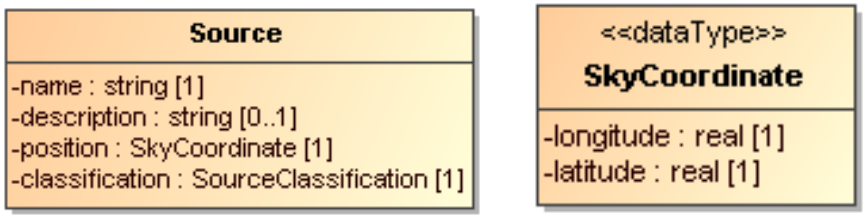


Figure 5 The ObjectType Source and the DataType SkyCoordinate both define attributes. Attributes can represent a PrimitiveType ('name' and 'description' in Source, 'longitude' and 'latitude' in SkyCoordinate), an Enumeration ('classification' in Source), or a structured DataType ('position' in Source).

PrimitiveType attribute as FIELDref, Enumeration as PARAM:
In the indirect representation of the ObjectType below a FIELDref indicates that an attribute is stored in field with ID="_designation". It does so using the utype of the attribute: SAMPL:source/Source.name, identifiying the name attribute of the Source type. Note that in our example we use a utype syntax derived from the VO-DML model itself. From this string one can here infer directly that some role with name 'name' defined on a type named Source is represented. But that is all one might infer. In principle this could have been a string like 'src:123456789'. In both cases one would need to inspect the formal data model to find out precisely *what* kind of model element this utype represents.

Another attribute, ‘classification’, is represented by a PARAM through its utype value src:source.Source.classification and assigns it the value ‘galaxy’. The attribute has as datatype an Enumeration, SourceClassification and indeed the PARAM defines a VALUES element with various OPTIONs (note that that is almost useless for a PARAM that represents a single value directly anyway). Its set of Literals indeed contains a value ‘galaxy’. In general however, especially for existing “legacy” databases, one cannot expect that enumerated values will exactly correspond to those in a model. Some type of mapping might be required, however OPTION does not support this in VOTable (i.e. has no @utype attribute). The fact that this attribute is stored in a PARAM in the GROUP indicates also that all Source instances stored in the TABLE are classified as galaxies. A more realistic case would require the use of a FIELDref to assign a TD value to each instance (row) in the table.

```
<TABLE>
<GROUP utype="vo-dml:Instance.root" >
  <PARAM      name=”type”      utype=”vo-dml:Instance.type”      value=”src:source.Source”
datatype=”char” arraysize=”*”/>
  <FIELDref ref="_designation" utype=" vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <PARAM name=”type” utype=”src:source.Source.classification” value=”galaxy”>
  <VALUES><OPTION value=”galaxy”/><OPTION value=”star”/>...</VALUES></PARAM>
  <GROUP utype="src:source.Source.position">
    <PARAM   name=”type”   utype=”vo-dml:Instance.type”   value=”src:source.SkyCoordinate”
datatype=”char” arraysize=”*”/>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD></TR>
...
</TABLE>
```

DataType attribute as GROUP:
As DataType-s are structured, their natural representation in a VOTable is as a GROUP, whether used directly or indirectly. Hence if an attribute is defined in a VO-DML data model as representing a DataType, it is most naturally represented by a GROUP embedded in the GROUP of the structured type owning the attribute.

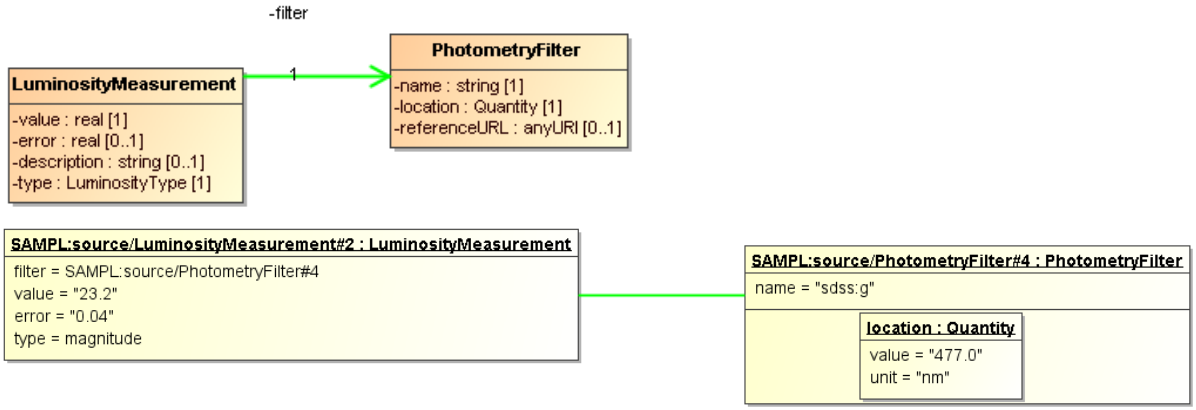
The example below shows in red a GROUP representing the attribute ‘position’ (identified by utype src:source.Source.position) that has as data type a SkyCoordinate, which itself consists of a ‘longitude’ and ‘latitude’ attribute (we defer discussing the reference to the next section). Note their structure indicates *only* their relation to their defining type, src:source.SkyCoordinate (though this, as discussed above, is unimportant for the current approach which, apart from the prefix, assigns no importance to the syntax of the utype identifiers in VO-DML models).

```
<TABLE>
<GROUP utype="vo-dml:Instance.root" >
  <PARAM      name=”type”      utype=”vo-dml:Instance.type”      value=”src:source.Source”
datatype=”char” arraysize=”*”/>
  <FIELDref ref="_designation" utype=" vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <PARAM name=”type” utype=”src:source.Source.classification” value=”galaxy”>
  <VALUES><OPTION value=”galaxy”/><OPTION value=”star”/>...</VALUES></PARAM>
```

```
  <GROUP utype="src:source.Source.position">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD></TR>
...
</TABLE>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD></TR>
...
</TABLE>
```

In this example the utype of the child GROUP *only* indicates its role in the definition of its parent GROUP, namely the attribute src:source.Source.position. It does not indicate the type, which is instead indicated by the PARAM with name “type”.

5.5 Mapping Reference



Referencing as ”GROUPref” to direct GROUP:
The example below uses a GROUP+@ref to represent the reference from a position object stored in a TABLE to a SkyCoordinateFrame stored in a direct GROUP. Hence all rows in the table use the same frame and the reference needs no structure.

```
<GROUP utype="vo-dml:Instance.root" ID="_icrs">
  <PARAM  name="type"  utype="vo-dml:Instance.type"  value="src:source.SkyCoordinateFrame"
datatype="char" arraysize="*"/>
  <PARAM utype="src:source.SkyCoordinateFrame.name" value="ICRS" .../>
  <PARAM utype="src:source.SkyCoordinateFrame.equinox" value="J2000.0" .../>
</GROUP>

<TABLE>
<GROUP utype="src:source.Instance.root" id="_source">
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="src:source.Source"
datatype="char" arraysize="*"/>
  <FIELDref ref="_designation" utype="vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <GROUP utype="src:source.Source.position">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
```

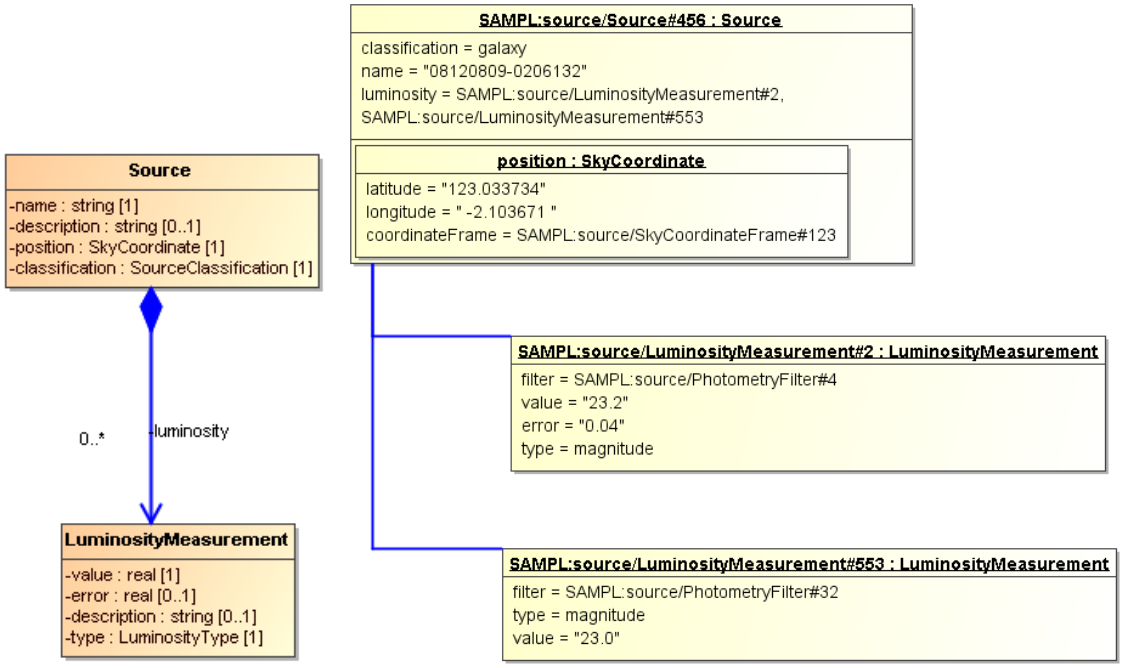
```
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
<FIELD id="_ designation " name="parentId" datatype="char"/>
<FIELD id="_ra" name="ra" datatype="float"/>
<FIELD id="_dec" name="dec" datatype="float"/>
...
<DATA><TABLEDATA>
<TR><TD>08120809-0206132</TD><TD>123.034</TD><TD>-2.1037</TD>...</TR>
...
</TABLEDATA></DATA>
</TABLE>
```

5.6 Mapping Collection

A collection is a composition relation between a parent ObjectType and a child, or *part*, ObjectType. The fact that objects can be stored in different ways implies many different ways in which the relation may need to be expressed. In XML serializations of a data model (such as used in VO-URP and the Simulation Data Model) one may choose to have the contained objects serialized *inside* the serialization of the parent. It is possible to do so in VOTable as well using child GROUPs representing a complete child object embedded in the GROUP representing the parent object. This is natural for this relationship because a collection element is really to be considered as a *part* of the parent object.

This may be used to represent flattening of the parent-child relation. One or more child objects may be stored together with the parent object in the same row in a TABLE. Such a case is actually very common, for example when interpreting tables in typical source catalogues. These generally contain information of a source together with one or more magnitudes. The latter can be seen as elements form a collection of photometry points contained by the sources.

Hence a GROUP inside another GROUP MAY represent a collection. It can do so in different modes that are illustrated by the following examples and described in more detail in section 0.



Container and contained objects in same row in same TABLE:
A very typical case is the following example, which shows a Source object serialized in the same row as its luminosities. The Source data model models luminosity measurements as a collection, which is flexible and allows measurements from different source to be combined in a natural manner. But for a given catalogue such as SDSS or 2MASS, it is known *a priori* how many magnitudes will be supplied and which bands they will correspond to. Hence for a given catalogue the natural representation is to simply add these as attributes to their model. Interestingly enough, the approach proposed here is able to support this mapping without any problem, even making a natural link to the actual photometry filter used for the measurements.

```
<TABLE>
<GROUP utype="vo-dml:Instance.root">
  <PARAM utype="vo-dml:Instance.type" value="src:source.Source" . . ./>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <GROUP utype="src:source.Source.position">
    <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
  <GROUP utype="src:source.Source.luminosities">
      <PARAM utype="vo-dml:Instance.type" value="vo-dml:Collection" . . ./>
      <GROUP utype="vo-dml:Instance.item”>
        <PARAM utype="vo-dml:Instance.type" value="src:source.LuminosityMeasurement" .
. ./>
        <FIELDref ref="_magJ" utype="src:source.LuminosityMeasurement.value"/>
        <FIELDref ref="_errJ" utype="src:source.LuminosityMeasurement.error"/>
        <GROUP ref="2massJ" utype="src:source.LuminosityMeasurement.filter"/>
      </GROUP>
      <GROUP utype="vo-dml:Instance.item”>
        <PARAM utype="vo-dml:Instance.type" value="src:source.LuminosityMeasurement" .
. ./>
        <FIELDref ref="_magK" utype="src:source.LuminosityMeasurement.value"/>
        <FIELDref ref="_errK" utype="src:source.LuminosityMeasurement.error"/>
```

```
        <GROUP ref="2massK" utype="src:source.LuminosityMeasurement.filter"/>
      </GROUP>
      <GROUP utype="vo-dml:Instance.item">
        <PARAM utype="vo-dml:Instance.type" value="src:source.LuminosityMeasurement" .
. ./>
        <FIELDref ref="_magH" utype="src:source.LuminosityMeasurement.value"/>
        <FIELDref ref="_errH" utype="src:source.LuminosityMeasurement.error"/>
        <GROUP ref="2massH" utype="src:source.LuminosityMeasurement.filter"/>
      </GROUP>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" utype="ivoa_1.0:stdtypes/string">
<DESCRIPTION>source designation formed from sexigesimal coordinates</DESCRIPTION>
</FIELD>
<FIELD name="ra" ID="_ra" unit="deg">
<DESCRIPTION>right ascension (J2000 decimal deg)</DESCRIPTION>
</FIELD>
<FIELD name="dec" ID="_dec"  unit="deg">
<DESCRIPTION>declination (J2000 decimal deg)</DESCRIPTION>
</FIELD>
<FIELD name="magJ" ID="_magJ">
<DESCRIPTION>J magnitude</DESCRIPTION>
</FIELD>
<FIELD name="errJ" ID="_errJ" unit="deg">
<DESCRIPTION>J magnitude error</DESCRIPTION>
</FIELD>
FIELD name="magH" ID="_magH">
<DESCRIPTION>H magnitude</DESCRIPTION>
</FIELD>
<FIELD name="errH" ID="_errH" unit="deg">
<DESCRIPTION>H magnitude error</DESCRIPTION>
</FIELD>
FIELD name="magK" ID="_magK">
<DESCRIPTION>K magnitude</DESCRIPTION>
</FIELD>
<FIELD name="errK" ID="_errK" unit="deg">
<DESCRIPTION>K magnitude error</DESCRIPTION>
</FIELD>
<TR>
<TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD><TD>23.2</TD><TD>.04</TD>
    <TD>23.0</TD><TD>.03</TD> <TD>23.5</TD><TD>.03</TD>
</TR>
...
</TABLE>
```

Container and collection all as direct instances in GROUPs:

When not using tables at all in a mapping, the individual elements form a collection can be directly represented inside the parent GROUP. To indicate this explicitly the utype attributes MUST be concatenations of the role (the collection) and the type.

```
<GROUP utype="src:source.Source.luminosities">
      <PARAM utype="vo-dml:Instance.type" value="vo-dml:Collection" . . ./>
      <GROUP utype="vo-dml:Instance.item”>
        <PARAM utype="vo-dml:Instance.type" value="src:source.LuminosityMeasurement" .
. ./>
        <PARAM value=”23.2” utype="src:source.LuminosityMeasurement.value" …/>
        <PARAM value=”.04” utype="src:source.LuminosityMeasurement.error" …/>
        <GROUP ref="2massJ" utype="src:source.LuminosityMeasurement.filter"/>
      </GROUP>
. . .
</GROUP>
```

5.7 Mapping value types

The examples above started from the assumption that the basis of a serialization was an ObjectType. Value types only show up as attributes. There may be some use cases however where one wishes to indicate *only* that a certain column or set of column represent some known value type. One reason may be that a standard, global data model does not exist that defines ObjectType-s matching the one in one's serialization, but that one can identify some sub-components that could be mapped to a DataType for example.

In fact the SourceDM used here is an example. It is a model for Source-s. The IVOA does currently not have an accepted model for this concept, though attempts in this direction have been made[6]. This implies many tables of interest in the VO can currently not formally declare they store instances of a Source. However they could declare they have columns that together correspond to a coordinate on the sky in the STC model. This could lead to a VOTable fragment as the following, which is a version of an example in 5.4, but with altered utypes, and removal of the GROUP representing the Source.

```
<TABLE>
<GROUP utype="vo-dml:Instance.root">
  <PARAM    name="type"    utype="vo-dml:Instance.type"    value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
  <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
  <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
  <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD></TR>
...
</TABLE>
```

Tools that understand STC may be able to do something with this annotation, even though they cannot know what role the coordinate plays.

Another example arises from a possible access protocol specification. Say Simple Cone Search (SCS) would declare that the result of a request MUST be a VOTable with a GROUP representing the actual request consisting of a position and a search radius. It could insist the position must be serialized using a GROUP representing an STC coordinate following our data modeling serialization prescription. E.g. as in the following example:

```
<GROUP utype="src:source.SkyCoordinateFrame" ID="_icrs">
  <PARAM utype="src:source.SkyCoordinateFrame.name" value="ICRS" .../>
  <PARAM utype="src:source.SkyCoordinateFrame.equinox" value="J2000.0" .../>
</GROUP>
...
<GROUP name="SCS">
  <INFO value="The SCS request "/>
  <PARAM name="SR" datatype="float" utype="ivoa_1.0:stdtypes/real"/>
  <GROUP name="center" utype="vo-dml:Instance.role">
    <PARAM    name="type"    utype="vo-dml:Instance.type"    value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <INFO value="The center coordinate of the simple cone search"/>
    <PARAM    name="ra"    utype="src:source.SkyCoordinate.longitude"    value="123.00000"
datatype="float"/>
    <PARAM    name="dec"    utype="src:source.SkyCoordinate.latitude"    value="-2.10000"
datatype="float"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
```

[6] See http://wiki.ivoa.net/twiki/bin/view/IVOA/IVAODMCatalogsWP.

```
  </GROUP>
</GROUP>
...
```

Note, this example even assigns a utype for the search radius SR, identifying it as the PrimitiveType “ivoa_1.0:stdtypes/real”. This shows that also PrimitiveType-s and Enumerations could be used directly, i.e. outside of a role they play.
One could argue that alternatively a standard protocol like SCS might define a little standard data model to represent its full request and use it in the serialization of result. In that case also the parent group, currently named “SCS’’, could have been declared to represent say an “scs:Request”, as follows:

```
 <GROUP utype="src:source.SkyCoordinateFrame" ID="_icrs">
  <PARAM utype="src:source.SkyCoordinateFrame.name" value="ICRS" .../>
  <PARAM utype="src:source.SkyCoordinateFrame.equinox" value="J2000.0" .../>
</GROUP>
...
<GROUP name="SCS" utype="vo-dml:Instance.root">
  <PARAM  name="type"  utype="vo-dml:Instance.type"  value="scs:Request"  datatype="char"
arraysize="*"/>
  <INFO value="The SCS request"/>
  <PARAM name="SR" datatype="float" utype="scs:Request.SR "/>
  <GROUP name="center" utype="scs:Request.center">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <INFO value="The center coordinate of the simple cone search"/>
    <PARAM    name="ra"    utype="src:source.SkyCoordinate.longitude"    value="123.00000"
datatype="float"/>
    <PARAM    name="dec"    utype="src:source.SkyCoordinate.latitude"    value="-2.10000"
datatype="float"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
...
```

5.8 Mapping Inheritance

We have introduced the Source Data Model to provide concrete mapping examples. Let’s now assume that a Data Provider has some information that pertains to astronomical sources, such as the redshift of the source, and they want to serialize this information in their files. The Source class in Source DM does not provide such a field. The solution is for the Data Provider to *extend* the Source class in Source DM.

Notice that the Data Provider might as well decide to include the information in a customized fashion, for example setting a particular name for the redshift column. However, this customized annotation requires readers to know the specifics of each custom file and their annotation. Instead, by adopting a common framework and this mapping language, the information can be provided to the user interactively and consistently, even by model-agnostic clients.

It is important that naïve clients find the information about the parent class without needing to parse anything but the VOTable. Advanced applications must be able to provide the information about the child class fields dynamically, while provider specific libraries can be derived from standard libraries to implement new functions.

First of all, the Data Provider must include a VO-DML declaration pointing to the description file and introducing a custom prefix:

```
<GROUP utype="vo-dml:Model" name="Source">
   <PARAM     utype="vo-dml:Model.url"     name="url"     datatype="char"     arraysize="*"
value="https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/sample/Sample.vo-dml.xml" />
   <PARAM    utype="vo-dml:Model.prefix"    value="src"    name="prefix"    datatype="char"
arraysize="*" />
</GROUP>
<GROUP utype="vo-dml:Model" name="ExtendedSource">
   <PARAM     utype="vo-dml:Model.url"     name="url"     datatype="char"     arraysize="*"
value="https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/sample/ExtendedSource.vo-dml.xml" />
   <PARAM     utype="vo-dml:Model.uri"     name="uri"     datatype="char"     arraysize="*"
value="ivo://ivoa.net/std/ExtendedSourceDM" />
   <PARAM    utype="vo-dml:Model.prefix"    value="xsrc"    name="prefix"    datatype="char"
arraysize="*" />
</GROUP>
```

In the description document the new field might have the ID ‘source.Source.redshift’, which translate to the utype ‘xsrc:source.Source.redshift’. This utype can be used to annotate instances of the extended class, like in the following example:

```
<TABLE>
<GROUP utype="vo-dml:Instance.root" >
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="xsrc:source.Source"
datatype="char" arraysize="*"/>
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="src:source.Source"
datatype="char" arraysize="*"/>
  <FIELDref ref="_designation" utype=" vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <FIELDref ref="_z" utype="xsrc:source.Source.redshift"/>
  <PARAM name="type" utype="src:source.Source.classification" value="galaxy">
  <GROUP utype="src:source.Source.position">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<FIELD name="z" ID="_z" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-2.103671</TD><TD>0.123</TD></TR>
...
</TABLE>
```

Notice that there are now two PARAMs with the vo-dml:Instance.type utype. This explicit mechanism allows clients to easily discover what types a GROUP serializes. There MUST be a PARAM for each class in the hierarchy. Multiple inheritance is supported, since there are no actual methods to implement/inherit apart from the getters and setters of the fields, for which there is no ambiguity. Another way to see this is that the types are actually interfaces, and even languages like Java, which do not support multiple inheritance, allow classes to implement more than one interface.

So, the extending class inherits all of the parent’s attributes **and their UTYPEs**: this way a naïve client of SourceDM can find all the src: elements, while a client of the child class can **also** look for the xsrc: elements. The knowledge of the Data Model, either hardwired

in the reader or dynamically generated using the VO-DML XML description, provides the client with the ability to look for the UTYPEs it is interested in.

6 Patterns for annotating VOTable: specification

In this section we list all legal mappings where a VOTable element uses its @utype to identify a VO-DML elements; we describe how such an annotation should be interpreted and what restrictions there are on the association. In its subsections it explicitly lists possible annotations of a VOTable with a UTYPE value that one may encounter following this spec.

The organization of the following sections is based on the different VO-DML concepts that can be represented. Each of these subsections contains sub-subsections which represent the different possible ways the concept may be encountered in a VOTable and discuss rules and constraints on those annotations. We start with Model, and then we discuss value types (PrimitiveType, Enumeration and DataType) and Attributes. Then ObjectType and the relationships, Collection, Reference and Inheritance (extends). Package is not mapped: none of the use cases required this element to be actually mapped to a VOTable instance.

Each subsection contains a concise, formal description of the mapping, according to a simple "grammar": Data Model elements are enclosed in square brackets, inside of which we describe the mapping pattern from the VO-DML element on the left to the VOTable element on the right. The mapping is realized by the elements appearing after the "/". A possible context is indicated by a ∈ with to its right a container formatted like the element of the left. Finally, a '→' indicates a relation to another element: this will generally be defined in the same VOTable, but it might be serialized elsewhere (e.g. when referencing a PhotometryFilter in a photometry catalog, whereas the actual serialization of the filter is in a different document).

For example the pattern expression "[GROUP ⇒ ObjectType / @utype="vo-dml:Instance.type]" can be read like this: "A GROUP *realizes* an ObjectType element *with* a utype attribute whose value is exactly 'vodml:Instance.type'"; similarly "[GROUP ⇒ ObjectType] ∈ RESOURCE" represents the same case restricted to GROUPs defined directly on a RESOURCE, i.e. *not* on a TABLE or another GROUP.

Appendix A contains the list of all the legal mappings.

Some comments on how we refer to VOTable and VO-DML elements

- When referring to VOTable elements we will use the notation by which these elements will occur in VOTable documents, i.e. in general "all caps", E.g. GROUP, FIELD, (though FIELDref).
- When referring to rows in a TABLE element in a VOTable, we will use TR, when referring to individual cells, TD. Even though such elements only appear in the TABLEDATA serialization of a TABLE. When referring to a column in the TABLE we will use FIELD, also if we do not intend the actual FIELD element annotating the column.
- When referring to an XML attribute on a VOTable element we will prefix it with a '@', e.g. @utype, @ref.

- References to VO-DML elements will be capitalized, using their VO-DML/XSD type definitions. E.g. ObjectType, Attribute.
- Some mapping solutions require a reference to a GROUP elsewhere in the VOTable. We refer to such a construct as a "GROUPref", which is not an element of the current VOTable standard (v1.3). So we use a GROUP with a @ref attribute, which must *always* identify another GROUP in the same document. The target GROUP must have an @id attribute. In cases where this is important we will indicate that this combination is to be interpreted as a "GROUPref", including the quotes.

The following list defines some shorthand phrases (underlined), which we use in the descriptions below:

- Generally when using the phrase type we mean a "kind of" type as defined in VO-DML. These are PrimitiveType, Enumeration, DataType and ObjectType.
- With atomic type we will mean a PrimitiveType or an Enumeration as defined in VO-DML.
- A structured type will refer to an ObjectType or DataType as defined in VO-DML.
- With a property available on or defined on a (structured) type we will mean an Attribute or Reference, or (in the case of ObjectTypes) a Collection defined on that type itself, or inherited from one of its base class ancestors.
- A VO-DML type plays a role in the definition of another (structured) type if the former is the declared data type of a property available on the latter.
- When writing that a VOTable element represents a certain VO-DML type, we mean that the VOTable element is mapped either directly to the type, or that it identifies a role played by the type in another type's definition.
- A descendant of a VOTable element is contained in that element, or in a descendant of that element. This is a standard recursive definition and can go up the hierarchy as well: an ancestor of an element is the direct container of that element, or an ancestor of that container.

6.1 Model

6.1.1 Model to GROUP in VOTABLE

Pattern expression: [GROUP ⇒ Model / @utype="vo-dml:Model"] ∈ VOTABLE

A GROUP element with @utype attribute identifying a Model and placed directly under the root VOTABLE element indicates that the corresponding VO-DML model is used in @utype associations.

Restrictions

- GROUP element must exist directly under VOTABLE and have @utype="vo-dml:Model"
- MUST have child PARAM element with @utype="vo-dml:Model.uri" and @value the URI of the VO-DML document representing the model. @name is irrelevant, @datatype="char" and arraysize="*". This annotation allows clients to *discover* whether a particular model is used in the document, the prefix of its @utype in the document,

and to resolve the Model to its VO-DML description. The URI MUST be a IVORN registered in the VO registries.

- SHOULD have child PARAM element with @utype="vo-dml:Model.url" and @value the url of the VO-DML document representing the model. @name is irrelevant, @datatype="char" and arraysize="*". This is a convenient shortcut for the resolution of the URI. Data providers should make sure the URL is not broken. Clients should make sure that they fall back to resolving the URI if the URL is broken.
- MUST have child PARAM element with @utype="vo-dml:Model.name" and @value the name of the Model, which also works ad the UTYPE prefix (see Section 4). @name is irrelevant, @datatype="char" and arraysize="*".

Example

```
<VOTABLE>
<GROUP utype="vo-dml:Model" name="Sample">
  <PARAM     utype="vo-dml:Model.url"     name="url"     datatype="char"     arraysize="*"
value="http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/sample/Source.vo-
dml.xml"/>
  <PARAM     utype="vo-dml:Model.uri"     name="uri"     datatype="char"     arraysize="*"
value="ivo://ivoa.net/SourceDM"/>
  <PARAM    utype="vo-dml:Model.prefix"    name="prefix"    datatype="char"    arraysize="*"
value="SAMPL"/>
</GROUP>
<GROUP utype="vo-dml:Model" name="IVOA_Profile">
<PARAM      utype="vo-dml:Model.url"      name="url"      datatype="char"      arraysize="*"
value="http://volute.googlecode.com/svn/trunk/projects/dm/vo-dml/models/profile/IVOA.vo-
dml.xml"/>
<PARAM    utype="vo-dml:Model.prefix"    value="ivoa_1.0"    name="prefix"    datatype="char"
arraysize="*" />
</GROUP>
...
```

6.2 DataType

A DataType instance (also a *value*) is structured; it consists of values assigned to each of its attributes and possibly references. To represent the complete instance of a DataType the various attributes (and references) must be grouped together; in VOTable this is done using a GROUP element. GROUPs in fact can play two different roles, depending on where the instance's data is really stored. If all values are eventually stored exclusively in PARAMs in the GROUP, or possibly outside the GROUP but accessed through PARAMrefs, the GROUP *directly* represents a complete instance.

If even only one of the attributes is stored in a FIELD and accessed through a FIELDref, the GROUP *indirectly* represents possibly multiple instances, one for each TR. This is also true in any child GROUP containing a FIELDref and so on. We will use these terms, *direct* and *indirect* representation all through the document. And note that this same classification holds for ObjectTypes discussed in 6.4, though with a twist related to possible child GROUPs representing Collections.

The attribute values of a DataType are stored according to the prescription for storing instances of their data type. For attributes with declared data type a PrimitiveType or Enumeration section provides details. If the attribute's data type is itself a DataType the prescription in the current section should be used recursively. DataTypes can also have References to ObjectTypes, but a discussion of how to store References is deferred to

section 6.5, after the discussion of storing ObjectType instances, which is provided in section 6.4.

A GROUP @utype never identifies a DataType directly. The DataType is identified by a PARAM with @utype vo-dml:Instance.type. The value of the PARAM is a UTYPE itself, referencing the DataType.

The GROUP @utype always identifies the role of the instance serialized by the GROUP, according to the Data Model. However, if the GROUP represents an instance that does not have a role, then the @utype MUST identify one of the following default Attributes:

- vo-dml:Instance.root – if the GROUP represents a root element
- vo-dml:Instance.attribute – if the GROUP is contained in a GROUP that does not identify any VO-DML type.
- vo-dml:Collection.item – if the GROUP represents an element that is part of a collection

6.2.1 DataType to GROUP in RESOURCE

Pattern expression: [GROUP ⇒ DataType / PARAM(@utype="vo-dml:Instance.type")] ∈ RESOURCE

A GROUP in a RESOURCE cannot have FIELDrefs, only PARAMs, PARAMrefs and GROUPs. The GROUP represents therefore a single instance of the DataType directly as defined in 7.2, with the PARAMs etc. providing the values of the components of the DataType.

The possible role this instance plays in the VOTable document must have been defined outside of any mapping to a data model. After all, in contrast to instances of ObjectTypes, the existence of instances of value types need not be explicitly stated (see the description of value types in section TBD). An example could be a VOTable containing the result of a simple cone search. A RESOURCE in that document might contain a GROUP representing the position of the cone, which may be mapped through its @utype to a DataType "src:source.SkyCoordinate" (if such a type existed: our sample model contains a type like it).

Clients MUST NOT assume that there is only one instance of the "vo-dml:Instance.type" PARAM in a GROUP

Example: A GROUP defined as a child of RESOURCE

```
<RESOURCE>
<GROUP utype="vo-dml:Instance.root">
  <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
  <INFO value="The center coordinate of the simple cone search"/>
  <PARAM name="ra" utype="src:source.SkyCoordinate.longitude" value="123.00000" . . ./>
  <PARAM name="dec" utype="src:source.SkyCoordinate.latitude" dec=-2.10000 . . ./>
  <PARAMref ref="_icrs" utype="src:SkyCoordinate.frame" . . ./>
</GROUP>
...
```

Example: A GROUP defined as a child of a GROUP with no UTYPE

```
<RESOURCE>
```

```
<GROUP>
 <GROUP utype="vo-dml:Instance.attribute">
  <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
  <INFO value="The center coordinate of the simple cone search"/>
  <PARAM name="ra" utype="src:source.SkyCoordinate.longitude" value="123.00000" . . ./>
  <PARAM name="dec" utype="src:source.SkyCoordinate.latitude" dec=-2.10000 . . ./>
  <PARAMref ref="_icrs" utype="src:SkyCoordinate.frame" . . ./>
 </GROUP>
...
```

6.2.2 DataType to GROUP in TABLE

Pattern expression: [GROUP ⇒ DataType / PARAM(@utype="vo-dml:Instance.type")] ∈ TABLE

A GROUP defined on a TABLE, annotated with a DataType, represents a DataType instance (directly or indirectly. In the direct case we can then interpret the GROUP "merely" as a structured PARAM following the VOTable spec [REF] "A PARAM may be viewed as a FIELD which keeps a *constant value* over all the rows of a table". I.e. a GROUP without FIELDref is a structured set of columns all with the same value in each row. The GROUP @utype may refer to an attribute in a DM, or not.

Example

```
<TABLE>
  <GROUP utype="vo-dml:Instance.root">
    <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
    <FIELDref ref="_ra" utype="bSTC:SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="bSTC:SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="bSTC:SkyCoordinate.frame"/>
  </GROUP>
...
<FIELD name="ra" ID="_ra" unit="deg">
<DESCRIPTION>right ascension (J2000 decimal deg)</DESCRIPTION>
</FIELD>
<FIELD name="dec" ID="_dec"  unit="deg">
<DESCRIPTION>declination (J2000 decimal deg)</DESCRIPTION>
</FIELD>
...
```

6.2.3 DataType to GROUP in a GROUP with no @utype

Pattern expression: [GROUP ⇒ DataType] ∈ [GROUP / ¬@utype]

We explicitly mention here a case already defined implicitly in the previous sections. Notice that by "no @utype" we mean either the case where the @utype is missing or the case where the @utype has a prefix not declared in the VO-DML preamble.

A GROUP may be defined inside another GROUP and be mapped to a DataType. There are some restrictions. NONE of its ancestor GROUPs MAY be mapped to a data model element. This would conflict with rules of mapping such an ancestor GROUP that state that children of such GROUPs MUST be mapped to properties of the structured type represented by the containing GROUP (see the restrictions listed in the following sections).

Whether the GROUP represents the DataType instance directly or indirectly is independent of this embedding in a parent GROUP, only of the existence of a FIELDref in it or its descendants.

Example A GROUP representing parameters of a Simple Cone Search

```
<GROUP name="ra"">
...
<GROUP utype="vo-dml:Instance.attribute">
  <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
  <INFO value="The center coordinate of the simple cone search"/>
  <PARAM name="ra" utype="src:source.SkyCoordinate.longitude" value="123.00000" . . ./>
  <PARAM name="dec" utype="src:source.SkyCoordinate.latitude" dec=-2.10000 . . ./>
  <PARAMref ref="_icrs" utype="src:SkyCoordinate.frame" . . ./>
 </GROUP>
</GROUP>
```

6.3 Attribute

An attribute is the role a value type plays in the definition of a structured type. For instance, 'longitude' is an Attribute of the 'SkyCoordinate' type, and 'position' is an Attribute of the 'Source' type.
An Attribute also has a type, including, in many cases, (structured) DataTypes or ObjectTypes.
The element representing the attribute MUST be contained in a GROUP that represents the containing structured type.

6.3.1 Attribute to FIELDref in GROUP

Pattern expression: [FIELDref ⇒ Attribute / @utype = AttributeID & @ref = FIELD-ID] ∈ [GROUP ⇒ ⟨ObjectType | DataType⟩]

A FIELDref contained in a GROUP MUST declare the Attribute serialized by the referred FIELD.

Restrictions

- The Attribute MUST be available to the structured type represented by the containing GROUP.
- The Attribute MUST have an atomic type.
- The datatype of the referenced FIELD must be compatible with the declared datatype of the Attribute.

Example
See example in 5.4.

6.3.2 Attribute to PARAM in GROUP

Pattern Expression: [PARAM ⇒ Attribute / @utype = AttributeID] ∈ [GROUP ⇒ ⟨ObjectType | DataType⟩]

When defined inside of a GROUP that represents a structured type, a PARAM MAY represent an Attribute of the type itself. The Attribute is declared by the @utype.

Restrictions

- The Attribute must have an atomic data type.
- The PARAM @datatype SHOULD be a compatible, valid serialization type for the type of the Attribute.

Example
See example in **Error! Reference source not found.**.

6.3.3 Attribute to PARAMref in GROUP

Pattern expression: [PARAMref ⇒ Attribute / @utype = AttributeID & @ref = PARAM-ID] ∈ [GROUP ⇒ ⟨ObjectType | DataType⟩]

Inside a GROUP a PARAMref identifies a PARAM defined inside the same RESOURCE or TABLE where the GROUP is defined. Using its @utype the PARAMref can annotate the PARAM as holding the value of an Attribute.

Restrictions

- The Attribute must be available to the structured type represented by the containing GROUP.
- Attribute's data type must be atomic.
- The PARAM must obey the restrictions defined for the annotation of a PARAM by the Attribute's type

Example
See example in 5.4

6.3.4 Attribute to GROUP in GROUP

Pattern Expression: [GROUP ⇒ DataType / @utype = AttributeID & PARAM(@utype="vo-dml:Instance.type")] ∈ [GROUP ⇒ ⟨ObjectType | DataType⟩]

A GROUP representing a structured type can have Attributes that are of a structured type as well.

This can be simply achieved by nesting a GROUP built according to Section
. However, this time the nested GROUP has a @utype pointing to the Attribute that the GROUP represents.

Restrictions

- Same as

Example

```
<GROUP utype="vo-dml:Instance.root">
  <PARAM utype="vo-dml:Instance.type" value="src:source.Source" . . ./>
```

```
  ...
  <GROUP utype="src:source.Source.position">
    <PARAM utype="vo-dml:Instance.type" value="src:source.SkyCoordinate" . . ./>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
...
</GROUP>
```

6.4 ObjectType

The patterns described for DataType apply to ObjectType as well. More patterns that apply only to ObjectType are described in the following sections.

Notice that there is a formal difference in VO-DML between DataType and ObjectType. From a practical point of view, these differences can be summarized as follows:

i) DataType does not inherit the vo-dml:ObjectType.ID attribute

ii) You cannot Reference a DataType instance

iii) You cannot have a Collection of DataType instances

[GL thinks that we should disallow “GROUPrefs” for DataTypes. However, it would then just be consistent to disallow PARAMrefs as well.]

[OL thinks that if we allow PARAMrefs than we should also allow “GROUPrefs” for DataTypes. In any case the “GROUPref” should not be considered part of the mapping, but an implementation detail: VOTable parsers should be able to follow VOTable references anyway.]

6.5 Reference

A Reference is a relation between a structured type (the “referrer”, an ObjectType or DataType) and an ObjectType, the “target object”, or “referenced object”. The reference is a property of the referrer: many referrers can reference the same target object.

6.5.1 Reference (from Object|DataType to ObjectType) to GROUP

Pattern Expression: [GROUP ⇒ Reference / @ref=GROUP-ID] ∈ [GROUP ⇒ ObjectType] → [GROUP ⇒ ObjectType / @id=GROUP-ID]

This pattern enables using References to other instances stored in the same document.

[TBD External references shall be treated in the ORM section]

A GROUP representing a structured type can refer to another ObjectType serialized in the same document by using the VOTable @ref -> @id mechanism.

We have to assume that each referenced object has an identifier and each referrer must somehow have a copy of that identifier. As the definition of identifiers is not exactly prescribed, neither can we prescribe the form that the reference will take.

Restrictions

- GROUP identified by @ref MUST represent an ObjectType compatible with the data type of the Reference, i.e. either the same type of some of its subtypes.

Example

```
<GROUP utype="phot:PhotometryFilter" ID="_2massJ">
  <PARAM    name="name"    datatype="char"    utype="phot:PhotometryFilter.name"
value="2mass:J"/>
...
</GROUP>
...
  <GROUP utype="src:source/Source.luminosity”>
  <PARAM utype="vo-dml:Instance.type" value="src:source/LuminosityMeasurement" ...
/>
    <FIELDref ref="_magJ" utype="SAMPL:source/LuminosityMeasurement.value"/>
    <FIELDref ref="_errJ" utype="SAMPL:source/LuminosityMeasurement.error"/>
    <GROUP  ref=”_2massJ” utype=”src:source/LuminosityMeasurement.filter”/>
  </GROUP>
...
```

6.5.2 Reference (from Object|DataType to ObjectType) to TR

Pattern expression: [GROUP ⇒ Reference / PARAM(@utype=”vo-dml:Instance.type”, @value=”vo-dml:Reference”) & PARAM(@utype=”vo-dml:Reference.id”)] ∈ [GROUP ⇒ ObjectType] → [GROUP ⇒ ObjectType / FIELDref(@utype=”vo-dml:ObjectType.ID)]

A GROUP representing a structured type can refer to a particular instance of an ObjectType indirectly represented by a GROUP, i.e. to a row in a TABLE. For this to be possible, the referred GROUP must define the instance ID by using a FIELDref with @utype equals to vo-dml:ObjectType.ID.

Thus, the referrer must have a @utype referencing its role (or a default role), a PARAM defining the instance type with value “vo-dml:Reference” and a PARAM that actually contains the ID of the referred object, with @utype “vo-dml:Reference.id”.

6.6 Collection

A Collection is defined in VO-DML as a composition relation between a parent ObjectType and a child, or *part*, ObjectType.

In XML serializations of a data model one may choose to have the contained objects serialized *inside* the serialization of the parent. It is possible to do so in VOTable as well using child GROUPs representing a complete child object embedded in the GROUP representing the parent object. This is natural for this relationship because a collection element is really to be considered as a *part* of the parent object. Of course, the contained object (the *item*) can be a Reference to an ObjectType described elsewhere.

This may be used to represent flattening of the parent-child relation. One or more child objects may be stored together with the parent object in the same row in a TABLE. Such a case is actually very common, for example when interpreting tables in typical source catalogues. These generally contain information of a source together with one or more magnitudes. The latter can be seen as elements form a collection of photometry points contained by the sources.

Hence a GROUP inside another GROUP MAY represent a collection. It can do so in different modes that are described in the following subsections.

Restriction

- The parent GROUP must represent an ObjectType that can contain the Collection.

6.6.1 Collection.item to GROUP in GROUP

Pattern expression: [GROUP ⇒ Collection.item / @utype="vo-dml:Collection.item"] ∈ [GROUP ⇒ Collection / PARAM(@utype="vo-dml:Instance.type", @value="vo-dml:Collection")]

Contained group represents an object of the indicated ObjectType that is an element of the indicated Collection. The GROUP MUST be contained inside of a GROUP that represents an ObjectType and is a valid Container of the Collection.

Example

See example in 5.6

The items in the collection of the above example are all actually serialized in the collection. It is also possible to have the items be References to ObjectTypes serialized elsewhere, following the Reference pattern.

6.7 Extends, inheritance

Pattern expression: [GROUP ⇒ ObjectType ⇑ ObjectType / PARAM(@utype="vo-dml:Instance.type")]

In this case the ⇑ symbol means that the leftmost ObjectType *extends* the rightmost ObjectType, adding some Attributes to it. Attributes can be structured or unstructured, direct or indirect, recursively. This mapping pattern leverages the patterns introduced in the previous sections.

The extending type inherits all of the parent type's Attributes and their UTYPEs (including the prefixes), and adds its own Attributes and their UTYPEs (including the prefixes).

It is possible that types in a Model extend types in the same Model. In this case one can already tell them apart from the a priori knowledge of a Model, or by parsing the VO-DML description of the Model, and UTYPEs cannot clash by definition. So, the pattern described here only applies to inter-Model extensions.

The extending type also inherits all of the parent's ancestors, recursively. Thus, the serialization of the child type MUST include all the ancestors' declarations of vo-dml:Instance.type. This makes it possible for clients of any ancestor to recognize the instance and to correctly apply polymorphism.

For example, assuming that the ExtentedSource Model extends the Source Model (Attribute source.Source.redshift), and that the ExtendedExtendedSource Model extends

the ExtendedSource Model (Attribute source.Source.profile), a serialization will look like this:

Example

```
<TABLE>
<GROUP utype="vo-dml:Instance.root" >
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="xsrc:source.Source"
datatype="char" arraysize="*"/>
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="xxsrc:source.Source"
datatype="char" arraysize="*"/>
  <PARAM      name="type"      utype="vo-dml:Instance.type"      value="src:source.Source"
datatype="char" arraysize="*"/>
  <FIELDref ref="_designation" utype=" vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <FIELDref ref="_z" utype="xsrc:source.Source.redshift"/>
  <FIELDref ref="_profile" utype="xxsrc:source.Source.profile"/>
  <PARAM name="type" utype="src:source.Source.classification" value="galaxy">
  <GROUP utype="src:source.Source.position">
    <PARAM   name="type"   utype="vo-dml:Instance.type"   value="src:source.SkyCoordinate"
datatype="char" arraysize="*"/>
    <FIELDref ref="_ra" utype="src:source.SkyCoordinate.longitude"/>
    <FIELDref ref="_dec" utype="src:source.SkyCoordinate.latitude"/>
    <GROUP ref="_icrs" utype="src:source.SkyCoordinate.frame"/>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
<FIELD name="ra" ID="_ra" unit="deg" .../>
<FIELD name="dec" ID="_dec"  unit="deg" .../>
<FIELD name="z" ID="_z" .../>
<FIELD name="profile" ID="_profile" .../>
<TR><TD>08120809-0206132</TD><TD>123.033734</TD><TD>-
2.103671</TD><TD>0.123</TD><TD>devaucoulers</TD></TR>
...
</TABLE>
```

In this example we have assumed that the VO-DML preamble declared the xsrc prefix for the ExtendedSource Model and the xxsrc prefix for the ExtendedExtendedSource Model.

6.8 Value, Unit, UCD

Some DataTypes may have attribute names that can be naturally mapped to the VOTable PARAM and FIELD attributes. In this case, the following rules apply:

i) the value of the DataType's 'value' attribute is stored in the @value attribute of the PARAM, or the TD corresponding to the annotated FIELD.

ii) the value of the DataType's 'unit' attribute is represented by the @unit attribute of the PARAM, or the annotated FIELD.

iii) the value of the 'ucd' attribute is mapped to the @ucd attribute of the annotated FIELD or PARAM

TBD Extend this pattern to datatype, arraysize? ivoa:Quantity DM.

6.9 ORM Mapping Patterns [TBD]

Some use cases (e.g. in SimulationDM) require support for more subtle patterns that are similar to the standard strategies used to design and implement relational databases.

Other, simpler cases, require that some elements' values refer to other elements defined elsewhere in the same file or in other files: consider for example a photometry catalog (or an SED) in which each row contains a reference to the photometry filter used for a particular measurement.

In these cases, standard strategies of Object-Relational Mapping can be easily implemented using the VO-DML to VOTable mapping. However, the same problem can be solved with each of such strategies, opening the door to implementation challenges: in order to constrain these strategies to a reasonable subset that is both sufficient to cover the use cases and simple to implement, some analysis is required. Much of this work has been done by the UTYPEs Tiger Team, but was not mature enough to be included in this draft.

7 Notable absences

The VOTable schema allows for redundancy in meta-data assignment. For example it allows assigning a UCD or UTYPE to FIELDrefs, but also to the FIELD it references. How is one to interpret or use this? Our approach is to try to avoid this redundancy.

The design laid out in the previous sections focuses UTYPE assignments on the GROUP element and its components. The main reason is that in all but the most simplistic use cases we will not be able to void the use of GROUPs, and that at the same time they provide all functionality (and more) that TABLE and FIELD could provide. Choosing this approach implies client coders do not need to take the possibly conflicting assignments into account, they only need consider GROUPs.

Here we list a few possible assignments that we avoid, though they might seem valid.

7.1 Atomic Types: support for custom and legacy UTYPEs

It is worth stressing explicitly that some VOTable elements are not covered by this specification (e.g. TABLE, RESOURCE, INFO, FIELD, and standalone PARAM). Also, according to this specification some elements will be ignored in the de-serialization of DM instances if their UTYPEs do not have a prefix declared in the VO-DML preamble.

This is intentional, and its purpose is to achieve full backward compatibility of this standard with the current non-standardized usages, while enabling new, complex Data Models to be effectively serialized in a standardized way.

Atomic Types are types referenced by @utype attributes of FIELDs and standalone PARAMs (i.e. PARAMs not included in GROUPs). Usually these UTYPEs are path-like strings pointing to some implicit and unspecified meta-model in an atomic fashion. Such UTYPEs can happily coexist with UTYPEs used according to this specification.

Not only this means that this standard does not break any of the existing standards. Not only this means that it enables customized use of the @utype attribute in local implementations. This also means that in order to make a legacy VOTable file compliant with the new specs, Data Providers will only need, if willing to do so, to add some GROUP definitions to its header. Old clients of that file will still be able to parse them, while new generation clients will be able to perform their more advanced usage of the new specification.

7.2 Packages

No use cases require the serialization of Package instances in VOTable. Package names are encoded in the UTYPE syntax of VO-DML for avoiding name clashes when two classes in different packages of the same model have the same name (other than that, UTYPEs are effectively opawue). The use of '.' as the separator for the Package->Type relationship (e.g. source.Source) might cause name clashes when a package contains a Type and a Package with the same name. However, packages cannot be directly referenced by UTYPEs, so this is not an issue.

7.3 ObjectType to TABLE

Pattern expression: [TABLE ⇒ ObjectType]

There might be some cases where a TABLE could be said to represent a structured type completely, and where the TABLE @utype attribute could make that identification. However in probably most cases only part of the TABLE will correspond to the type, or multiple types (or instances of a type) will be stored inside a single row (e.g. photometry catalogs).

In all of these cases one can (and MUST) use one or more GROUP elements contained by the TABLE to make the precise assignment.

By leaving this mapping pattern out of the specs we do not lose any information content. Also, this way client code only has to deal with GROUPs, with no need to inspect the TABLE.

7.4 Attribute to FIELD|PARAM in TABLE

Pattern expression: [Field ⇒ Attribute] ∈ [TABLE ⇒ ObjectType]
Pattern expression: [PARAM ⇒ Attribute] ∈ [TABLE ⇒ ObjectType]

Assigning an Attribute to a FIELD would only make sense in the context of a structured container, which can only be TABLE. But as we propose not to use TABLE to represent a DM element directly, consequently FIELD cannot be an attribute. We use FIELDref for that. This also enables backward compatibility, since the FIELD @utype can follow custom or legacy mapping rules.

For the same reason that makes us avoid the assignment of Attribute to FIELD, we avoid assigning an attribute to a standalone PARAM, i.e. a PARAM *that is directly contained in a TABLE*. The context (TABLE) is not used to indicate the type containing the Attribute. For this element a PARAMref or a PARAM inside a GROUP is to be used. This enables backward compatibility.

8 Serializing to other file formats [TBD]

VOTable is expressive enough to allow the mapping patterns described in this specification. Other formats (notably FITS) cannot support such annotations. Notice, for instance, that a UTYPE can be used to annotate a column (keyword TUTYPn) but cannot be used as a keyword itself. A solution to this, which is even one of the

motivation behind the VOTable specification, is to wrap other format's tables with VOTable headers. Some FITS formats also allow a VOTable header to be included in an HDU, but this solution seems to be less portable, since some FITS readers do not support these files.

More discussion will be needed to address this issue.

Appendix A. List of all valid mapping patterns [TBD]

Appendix B. Growing complexity: naïve, advanced, and guru clients

This document defines a complete, unambiguous, standard specification that can be used to serialize and de-serialize instances of Data Model types. It was designed to be simple and straightforward to implement by Data Providers and by different kind of clients. We can classify clients in terms of how they parse the VOTable in order to harvest its content. Of course, in the real word such distinction is somewhat fuzzy, but this section tries and describe the different levels of usage of this specification.

Naïve clients
We say that a client is naïve if:

i) it does not parse the VO-DML description file
ii) it assumes the a priori knowledge of one or more Data Models
iii) it discovers information by looking for a set of predefined UTYPEs in the VOTable

In other terms, a naïve client has knowledge of the Data Model it is sensitive to, and simply discovers information useful to its own use cases by traversing the document, seeking the elements it needs by looking at their @utype attribute.

Examples of such clients are the DAL service clients that allow users to discover and fetch datasets. They will just inspect the response of a service and present the user with a subset of its metadata. They do not *reason* on the content, and they are not interested in the structure of the serialized objects.

If such clients allow users to download the files that they load into memory, they should make sure to preserve the structure of the metadata, so to be interoperable with other applications that might ingest the same file at a later stage.

Advanced clients
We say that a client is advanced if:

i) it does not parse the VO-DML description file
ii) it is interested in the structure of the serialized instances
iii) can follow the mapping patterns defined in this specification, for example collections, references, and inheritance

Examples of such clients are science applications that display information to the user in a structured way (e.g. by plotting it, or by displaying its metadata in a user-friendly format), that *reason* on the serialized instances, perform operations on those instances, and possibly allow the users to save the manipulated version of the serialization.

Notice that the fact that an application does not directly use some elements that are out of the scope of its requirements does not mean that the application cannot provide them to the user in a useful way. For example, an application might allow users to build Boolean filters on a table, using a user-friendly tree representing the whole metadata. This exposes all the metadata provided by the Data Provider in a way that might not be meaningful for the application, but that may be meaningful for the user.

Notice, also, that advanced clients *may* be DM-agnostic: for instance, an Advanced Data Discovery application may allow the user to filter the results of a query by using a structured view of its metadata, even though it does not possess any knowledge of Data Models.

Guru clients
We say that a client falls into this category if:

i) it parses the VO-DML descriptions

ii) it does not assume any a priori knowledge of any Data Models.

Such applications can, for example, dynamically allow users and Data Providers to map their files or databases to the IVOA Data Models in order to make them compliant, or display the content of any file annotated according to this standard.

This specification allows the creation of universal validators equivalent to the XML/XSD ones.

It also allows the creation of VO-enabled frameworks and universal libraries. For instance, a Python universal I/O library can parse any VOTable according to the Data Models it uses, and dynamically build objects on the fly, so that users can directly interact with those objects or use them in their scripts or in science applications, and then save the results in a VO-compliant format.

Java guru clients could automatically generate interfaces for representing Data Models and dynamically implement those interfaces at runtime, maybe building different views of the same file in different contexts.

Notice that Guru frameworks and libraries can be used to build Advanced or even Naïve applications in a user-friendly way, abstracting the developers from the technical details of the standards and using first class scientific objects instead.

Appendix C. Frequently Asked Questions

Q: I was used to discover elements in a VOTable by trivially matching strings: can I still do it?
A: Yes, actually even more so: in fact, according to this specification, the very same string for the very same element will be present in any context that includes such element. For example, if you want to find the error bar of an element you can simply look for a UTYPE like Accuracy.StatError, which does not depend on whether the error refers

to a Flux or a Wavelength. Consider the real case of SPLAT, a science application that deals with spectra: in order to use the "old" UTYPEs, SPLAT cannot trivially match strings, but uses a regular expression (even though very simple) to match the *.Accuracy.* portion of a UTYPE. With the new specification in place, SPLAT could just trivially match strings by checking that they are equal.

Q: Will this specification increase the number of UTYPEs?
A: No. On the contrary, it will dramatically reduce them: consider for instance an Accuracy type, which basically encodes error bars. It is composed of few attributes that express symmetric and asymmetric error bars, systematic errors, upper and lower limits. In the new specification they can be univocally referenced by less than ten UTYPEs. In the "old" scheme, each of these elements was bound to other elements (e.g. the spectral axis, the flux axis, and so on): this means that for expressing the same concepts you needed more than one hundred UTYPEs only in the Spectrum Data Model!
Consider IRIS, an application that deals with SEDs and spectra: in this application about a thousand UTYPEs needed to be hard-coded in order to fully represent instances and provide a Java library for their I/O and manipulation. With this specification IRIS would need one order of magnitude less UTYPEs to be hard-coded, and could actually even make without them, by simply using the VO-DML description and a set of intelligently designed Java interfaces and objects.

Q: Will this specification increase the number of Data Models?
A: The simple answer is: No. This spec is agnostic about which data models there are. However, a whole lot of Data Models are already out there, since basically each astronomical instrument needs an ad-hoc Data Model. This specification allows such Data Models to be expressed in a standard, machine-readable format *in terms* of common, more generic Data Models defined and maintained by the IVOA. So, the extended answer is: No, but it will effectively increase the number of *interoperable* Data Models, allowing Data Providers to register their own Models in an IVOA registry.

Q: Am I now forced to use UML for using this specification?
A: No. However, part of the larger picture includes UML as a convenient tool to design, display, document, and register Data Models. The larger picture of which this specification is part includes a VO-UML specification in the form of a UML profile that can make the creation of Data Models more standard and easier. However, using VO-UML is not a requirement.

Q: Am I now forced to use specific software for using this specification?
A: No. However, this specification enables the development of software that can make the creation and use of Data Models easier. For example, such software might allow the automatic generation of documentation, code, web applications, and services from a simple set of UML diagrams. Some software was already created as part of this specification effort, or other IVOA efforts from which this specification was derived.

Q: Does this specification break the current standards and protocols?
A: No. This specification does not require any changes in services and applications that implement current IVOA standards and protocols. However, it also enables new, more complex standards to be designed and implemented. Files complying with old standards can be made compliant with this specification by simply adding metadata to them.

Q: Can I use UTYPEs in a customized way without violating this specification?

A: Yes. This specification explicitly allows custom usage of UTYPEs, under certain conditions (FIELDs, standalone PARAMs, etc.).

Q: It looks like this specification needs clients to recursively concatenate strings to build the actual UTYPE. Is this the case?
A: No. UTYPEs are defined so to be opaque strings that do not need to be parsed or to be concatenated in order to be used. They are simply references that map the VOTable elements to a standard “model of models”.

Q: This specification is so complicated! Do I need to implement it all, even just for extracting a flux from a VOTable?
A: No. And it is not that complicated, actually. As a DNA strand is a simple concatenation of four basic components, but it is used in nature to build complex organisms, this specification shows how simple identifiers can be used to enable complex scientific applications. How complex an implementation is depends on the implementation requirements: if they are simple, the specification enables you to meet them in a very simple way. For example, you can extract a flux by simply looking for a @utype attribute like src:source.LuminosityMeasurement.value. If the requirements for your application are more complex, then this specification makes the best effort to enable them straightforwardly. If you are a Data Provider and you want to publish compliant files and databases, the complexity of the task depends pretty much on the complexity of your files: they might be very simple structures of VOTable groups, or they could require complicated Object-Relational Mapping patterns. However, software can be built using this specification in order to make the data publishing process easier.

Q: Should I parse UTYPEs?
A: No. [It is still to be discussed whether prefixes are fixed or not. In the second case, some strategies, but not all, might require clients to strip the prefix off the ‘localname’ in order to look it up in the VODML/XML, in a way similar to what happens now, e.g. obs:Target.Name vd sdm:Target.Name.].