International

Virtual

Observatory

Alliance

Mapping Data Models to VOTable
Version 1.0-20150427
Working Draft 2015 April 27

This version:
1.0-20150427
Latest version:
1.0-20130509
Previous version(s):

Editors:
Gerard Lemson
Omar Laurino

Authors:
Gerard Lemson, Omar Laurino, Patrick Dowler, Makus Demleitner, Matthew Graham, 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. This specification defines a scheme for annotating VOTable instances in a standard, consistent, interoperable fashion, so that each piece of metadata can unambiguously refer to the correct Data Model element it expresses[1]. With this specification, data providers can unambiguously and completely represent Data Model instances in the VOTable format, and clients can build “faithful” representations of such instances. The mapping is operated through opaque, portable strings. These used to be called ‘utypes’,

[1] Assuming there is a suitable data model!

but document uses a new annotation mechanism. This proposal assumes agreement on a VOTable extension that adds an explicit mapping element <VODML> to selected VOTable elements (GROUP, PARAM, PARAMref, FIELDref) and which indicates that those elements are representing particular elements from some VO-DML model.

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/.

History of this document

This document and its sister VO-MDL specification [1] are a direct consequence from the efforts of the UTYPEs tiger team. The following are some of the main steps along the path to the current realization.

2012-08-11: First telecom of UTYPEs tiger team, creation of its IVOA wiki page , contains minutes of telecoms and links to various documents.
2012-10-09 document added on IVOA wiki with discussion on how VO-URP can support UTYPE-s discussion.
2013-02-19 First check-in to VOLUTE of discussion documents for mapping VO-DML to VOTable. *VO-DML and UTYPE and VOTable-v0.1.docx (+.pdf)* (volute/svn 1982). Version 0.2 (volute/svn 2079).
2013-04-12: Document split in two parts, planned for separate evolution through IVOA process, *VO-DML-WD-v0.x-20130412.doc* (title: *VO-DML: a proposal for a consistent modelling language for IVOA data models*) and *UTYPEs-WD-v0.x-201304-12.doc* (title: *UTYPEs: portable data model references*,) (volute/svn 2109)
2013-04-22 New version of utypes document *UTYPEs-WD-v0.5-20130422.docx*
2013-05-09. First stable v1.0 issue of utypes document released (volute/svn 2185).
2013-05-29. Title changed to *Mapping Data Models to VOTable*. Minor revision of abstract and introductions. (volute/svn 2830)
2015-04-27: Commit of version 1.0-20150427 (volute/...). Major reworking of whole document to take into account change from using utype attribute to new VODML element, addition of object-relational mapping and redesign of formal expression language for mapping patterns.

Table of Contents

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.

In order to enable such interoperable, extensible, portable annotation of data files, one needs:

i) A language to unambiguously and efficiently describe Data Models and their elements' identificators (VO-DML, [1]).
ii) Pointers linking a specific piece of information (data or metadata) to the Data Model element it represents[2].
iii) A mapping specification that unambiguously describes the mapping strategies that lead to faithful representations of Data Model instances in a specific format.

Without a consistent language for describing Data Models there can be no interoperability, both between them, reuse of models by models, or in their use in other specifications. Such a 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.

For serializations to non-standard representations 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. In particular several different interpretations of UTYPEs have been proposed and used[3]. This specification aims to resolve this ambiguity.

[2] This used to be the assumed role of the @utype attribute in VOTable and for example TAP. This document introduces the new <VODML> element for this purpose in VOTable, as agreed on in interop meeting in Banff, 2014.
[3] See the Current Usages Note:
http://www.ivoa.net/documents/Notes/UTypesUsage/20130213/NOTE-utypes-usage-1.0-20130213.pdf

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.

As a matter of fact, existing files and services can be made compliant according to this specification by simply *adding* annotations and keeping the old ones. So they do not need to *change* them in such a way that would necessarily make them incompatible with existing software.

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 specification describes how to represent Data Model instances using the VOTable schema. This representation uses the <VODML> element introduced for this purpose in VOTable v1.4 [TBC] 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.

In sections 1-6 we give an introduction to why and how the VODML elements can be used to hold pointers into the data models and several examples that illustrate the mapping.

Section 7 is a rigorous listing of all valid annotations, and the normative part of the specification. Section 8 describes what patterns and usages this specification does *not* 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. Section 9 described ideas how this specification might be used for annotating other tabular formats, and how to generalize it to other, more structured data serialization formats. Section 10 contains references.

The appendices contain additional material. Appendix A describes the VODML annotation element that was added to the VOTable schema to support this mapping specification. Appendix B describes different types of client software and how they could deal with VOTables annotated according to the current specification. Appendix C defines a set-based "language" for expressing mapping patterns in a more formal manner. Appendix D tries to answer some frequently asked questions.

Throughout the document we will refer to some real or example Data Models. Please remember that such models have been designed to be fairly simple, yet complex enough to illustrate all the possible constructs that this specification covers. They are not to 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 representing a specific concept. Given a VOTable annotated with VODML concepts, a client can extract a piece of information by finding a PARAM or FIELDref annotated with a predefined VODML/ROLE or VODML/TYPE. For example, the client can find the luminosity measurement(s) in a file by looking for the GROUP element containing a VODML/TYPE with value “src:source.LuminosityMeasurement”.

Serialize and de-serialize instances according to a data model. Using this specification (or software implementing it), a data provider can serialize the metadata for a dataset according to a data model. A client can build in memory a faithful representation of that instance according to the data provider’s annotations, assuming the knowledge of a finite set of VODMLREFs. 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.

Model-unaware serialization and de-serialization. Model-unaware readers and writers can serialize and de-serialize instances according to specific data models by mapping the contents of a VOTable to model description files. This may include file browsing, code generation, data integration, etc.

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* high-level 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 may help 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 may have non-compliant files that they want to convert to a VO-compliant format according to some data model: a DM-unaware Importer application may allow them to do so for any standard Data Model.

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 serialized 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[4].

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. We call such representations *faithful*.

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[5]. 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:

– 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

[4] See, for example, http://logic.stanford.edu/dataintegration/chapters/chap01.html
[5] Or alternatively as a transformation of a (standard) serialization of the data model.

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 9.

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.

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 Mapping with the <VODML> element.

Here we discuss the technicalities of *how* to annotate VOTable with VO-DML metadata. The later sections will provide details on the implied semantics of these annotations and rules on their application.
VOTable 1.2 introduced the @utype attribute, which was intended to represent "pointers into a data model". A precise and formal definition on how this “pointing” was to be achieved and a description of its meaning was missing though.
First, a formal definition of the target of the pointers was missing. To solve this, data models were usually accompanied by a list of “utypes” [TBD refer to STC, Characterization, Spectrum], and these could be used as values for said @utype, be it in VOTable or for example in the Table Access Protocol metadata. These were not linked in any formal, machine readable way to the underlying data model.
Basically it means that the data model is represented solely by a list of attributes, which does not do justice to the complexity of data models describing complex data products like Data Cubes or the provenance of Simulations. These contain complex object hierarchies organized in graphs with various types of relations between individual objects. It also proved difficult to express the relationship among different, but overlapping, data models, with much discussion centred on the question how to reuse utypes from one model in the definition of another.
The approach is basically not much more than another vocabulary, similar to UCDs [8], or SKOS vocabularies [11], obtained by different means. Efforts were made to provide some structure to these values that might provide some hints of their location in a model, but there was no formal mechanism on how to derive that structure[6] and it was unclear whether it could truly represent the richness of the existing and future data models. In

[6] Only for the simulation data model **Error! Reference source not found.**, which had a formal expression very similar to the one defined in the current document, was such a formalism defined.

particular there was no standard defined how this could be achieved and no common usage patterns were discovered [9].
As described above [TBD check, true?] VO-DML has solved the problem that there was no formal target for these pointers in the data model itself and formally defines how models can be reused in the definition of other, dependent models. Precisely *how* to use these pointers in a VOTable to provide a complete annotation useful for interoperability requires more work though. It requires restrictions on which VOTable elements can point to which concepts and how they are organized.

The current specification provides such a definition. It shows how data publishers can identify also the more complex data model elements such as structured types and relationships inside some published data source, be it a VOTable or relational database published through the TAP protocol. It notes that in fact the meta-model defined by the VOTable schema is perfectly suited for supporting such a *mapping* in an almost 1-1 and completely logical fashion.

This specification defines various *mapping patterns* from VOTable to VO-DML. Such a pattern identifies a VOTable element with a VO-DML element. The VO-DML element is said to be *represented* by the VOTable element. The mapping pattern indicates that instances of identified VO-DML types are present in the VOTable. These may be atomic *values* (instances of VO-DML ValueTypes [1]), represented by PARAM@value or by cells in a table column identified by a FIELDref. Alternatively they may be instances of structured types ([TBD ref to location of VO-DML doc]) represented by GROUPS consisting of multiple PARAMs, PARAMrefs, or FIELDrefs. Especially this use of the GROUP element, introduced already in version 1.10 of the VOTable specification, that distinguishes this approach from the previous ones. The GROUP element can be directly mapped to the structured types defied in VO-DM and by mapping its components to corresponding components in the VO-DML type definition a 1-1 mapping is achieved almost trivially[7]. The complete set of mapping patterns supported by this specification will be defined formally in section 7.

In the rest of this chapter we discuss *how* the VOTable elements are to identify the corresponding VO-DML element. Originally it was assumed that the *@utype* attribute of these VOTable elements (PARAM, PARAMref, FIELDref and GROUP) would be used to identify which data model element is represented. But it was decided *not* to use @utype after all[8]. Instead it was agreed that the VOTable schema would be extended with a new type elements, <VODML> to be added to these VOTable elements. This element would take the task that was originally intended for @utype, namely to point into a data model. The requirement of extending the VOTable schema is unfortunate and the existing schema would have sufficed to represent the mapping patterns[9], on the other it allows us to design a much more explicit annotation mechanism than would be possible when using @utype-s only.
The extension to the VOTable schema is reproduced in Appendix A. In the next few sub-sections we discuss this element in detail.

[7] This was foreseen for example in the note on referencing STC in VOTable (M. Demleitner, [10]), as well as in a presentation by S. Derriere in Naples:
(http://wiki.ivoa.net/internal/IVOA/InterOpMay2011SED/PPDMDesc_Naples.pdf).
Lacking a formal target modelling language these developments remained ad hoc.
[8] During various interoperability meetings, finalized in Banff 2014.
[9] See older versions of this document on volute, mentioned in the history above.

TBD see if we need next paragraphs here or in other parts of text

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 declared datatype 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 only annotating with the VO-DML **`Role`** may not be sufficient to infer all DM information about a VOTable element. Hence we include the possibility to annotate VOTable elements with the exact VO-DML **`Type`** as well.

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.

4.1 complexType: VODMLAnnotation

According to the schema fragment in Appendix A, the VODML element consists of elements <TYPE>, <ROLE> and possibly one or more <OPTION> elements. We ignore the latter element here, see the detailed definition in section 7 for their use.
A VODML/TYPE element MUST have as value a valid **`vodmlref`** (see next subsection) ,which formally indicates that the VOTable element it belongs to represents an instance of the identified VO-DML **`Type`**. The VO-DML/XML document that the **`vodmlref`** identifies through its prefix will give the formal definition of that type at the element identified by the suffix **`vodml-id`**.
Most instances of types are actually used in the definition of a larger, structured type: they play a role in its definition [TBD ref to location in VO-DML element]. In VOTable this structured parent type will be represented by a GROUP (see 7.2 or 7.2). A VOTable element that is contained in such a group MUST identify which precise role it represents through the VODML/ROLE element. This element is again a vodmlref that MUST identify a **`Role`** (i.e. **`Attribute`**, **`Reference`** or **`Composition`** relation) available on the **`Type`** identified by the parent GROUP.

These elements VODML/TYPE and VODML/ROLE

4.2 simpleType: VODMLReference

This type represents a reference to a single element in a VO-DML/XML document. It take over the role of the @utype attribute in this regards. Whenever we wish to refer to instances of the VODMLReference type we will call them **`vodmlref`**-s. A vodmlref is a string with the following syntax:

```
vodmlref ::= prefix  ‘:’ vodml-id.
```

The prefix identifies the model in which the element identified by the suffix is defined. How this is to be done is still under some discussion and we describe both versions next. [TBD version 1 is the currently accepted approach; needs rewrite once we settle on one of these]
`vodml-ids` are always considered opaque, meaning that clients have no reason to parse them. They are identifiers mapping VOTable elements to VO-DML elements in the identified data model. Thus, they must follow the same syntax rules defined in the VO-DML/Schema document.

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.

4.2.1 Version 1:

Prefixes MUST be exactly the same as the **`name`** attribute of the model in the VO-DML/XML document that defines it. They are sequences of [A-Za-z0-9_-], and they are case sensitive.
For new models, that are not (yet) standardized or for custom data models used in a smaller community, It is recommended to form DM prefixes as <author-acronym>_<dm-name>, where the <dm-name> is the name of a standard data model; thus, NED's derivation of spec could have ned_spec as a prefix, CDS's derivation cds_spec.

Prefixes correspond to major versions for the corresponding data models. Thus, **`vodmlrefs`** remain constant over "compatible" changes in the sense of [DOCSTD]. In consequence, clients must assume a compatible extension when encountering an unknown **`vodmlref`** 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 the GROUPs identifying which model sare used in a VOTable as described in section 7.1 below.

4.2.2 Version 2

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

4.2.2.1 The Namespace URI

The **vodmlref** prefix is a reference to a namespace URI defined in a VO-DML preamble (see 4.3, also 7.1).

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

4.2.2.2 How to look for a `vodmlref` in a document

Clients may match **vdmlrefs** by the simple string comparison of the <ROLE> or <TYPE> elementsin a VOTable with **vodml-ids** 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:vodml-id** syntax before it can be compared to the **vdmlrefs** in the document.

4.2.2.3 Extended notation for portable use: `vdmlrefs` as URIs

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

Thus, the extended notation for a **vdmlref** with **vodml-id** 'Foo.bar' will be:
ivo://authorityID/DM-ID#Foo.bar

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

This section assumes Alternative 1 in section 4.2
In order to signal to the reader that a VOTable document falls under this specification, a VOTable instance MUST declare all the Data Models it includes, their versions, and the **vodmlref** prefixes for this model. They MUST 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>
  <VODML><TYPE>vo-dml:Model</TYPE></VODML>
  <PARAM name="name" datatype="char" arraysize="*" value="src">
    <VODML><ROLE>vo-dml:Model.name</ROLE></VODML>
  </PARAM>
  <PARAM name=”version” datatype=”char” arraysize=”*” value=”1.0” >
    <VODML><ROLE>vo-dml:Model.version</ROLE></VODML>
  </PARAM>
  <PARAM name="url" datatype="char" arraysize="*"
      value="https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/sample/Source.vo-dml.xml" >
    <VODML><ROLE>vo-dml:Model.url</ROLE></VODML>
  </PARAM>
</GROUP>
```

The above example introduces the Source Data Model and declares the name “src” which is to be used as prefix to **`vodmlref`**-s pointing to its elementsIt also identifies the url referring to the VO-DML/XML description of the Data Model. Notice the “vo-dml:Model” special VODML/TYPE that annotates the GROUP element to introduce the declaration. No role should be used here.

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

Note that the VO-DML preamble MUST include the models corresponding to ALL the different prefixes used for the **`vodmlrefs`** in the VOTable. As a matter of fact, the preamble can be seen as a way to declare such prefixes and map them to models. Note furthermore that the **`vo-dml`** prefix used in the above annotations itself refers to an explicitly defined data model. It will be described in the next section.

4.4 Special annotations from the VO-DML Mapping model

There are some special **`vodmlrefs`** with prefix *vo-dml* [TBD should we use different name? E.g. vodml-i or so? To identify this models deals with mapping instances?] . These can be used to create specialized mapping patterns. We have already encountered some examples in the previous section to identify Models represented in a VOTable.
We actuall use the same mapping patterns defined in this specification for this. This implies we need a special data model, which we name *VO-DML Instance Mapping* and whose prefix is *vo-dml* [TBD should maybe change this to *vodml-i*]. The model can be found here[10].

5 General information about this spec

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.

[10] Note that this model does **not yet** obey all of the data modelling rules imposed on real data models. It is defined to keep up the invariant that each vodmlref MUST refer to an element in a formal VO-DML/XML element.

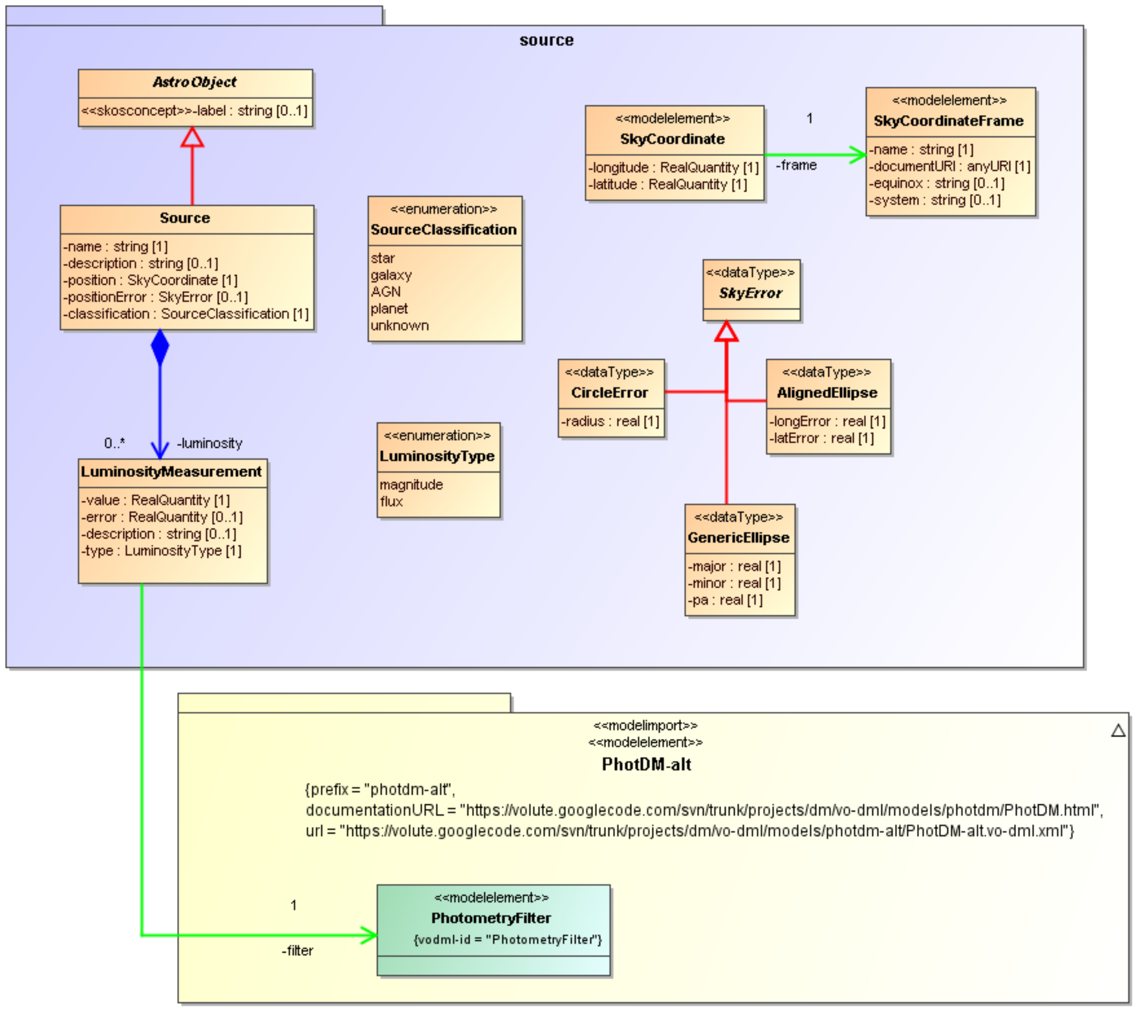


Figure 1 Data model used in examples. It represents a simplified Source data model, containing luminosities that refer to the imported PhotDM. It also defines a simplistic version of an STC model with some types for defining coordinates on the sky, for the sake of simplicity and just for example purposes.

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 `vodml-ids` we use to identify the different elements. The values we use for these `vodml-id` 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 indication of the location of the element they point to 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.

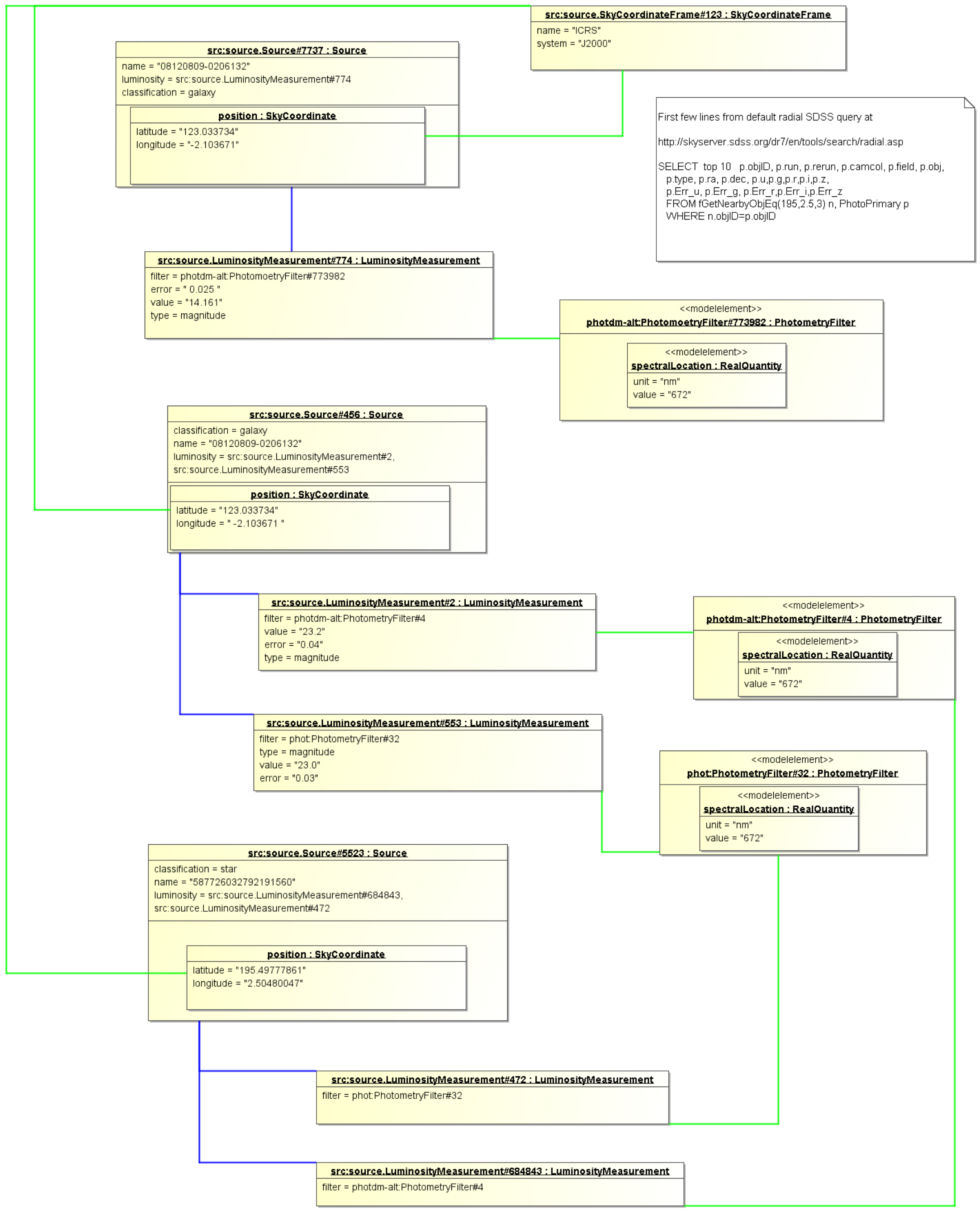

src:source.SkyCoordinateFrame#123 : SkyCoordinateFrame
name = "ICRS"
system = "J2000"
src:source.Source#7737 : Source
name = "08120809-0206132"
luminosity = src:source.LuminosityMeasurement#774
classification = galaxy
position : SkyCoordinate
latitude = "123.033734"
longitude = "-2.103671"
First few lines from default radial SDSS query at
http://skyserver.sdss.org/dr7/en/tools/search/radial.asp
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
src:source.LuminosityMeasurement#774 : LuminosityMeasurement
filter = photdm-alt:PhotomoetryFilter#773982
error = " 0.025 "
value = "14.161"
type = magnitude
<<modelelement>>
photdm-alt:PhotomoetryFilter#773982 : PhotometryFilter
<<modelelement>>
spectralLocation : RealQuantity
unit = "nm"
value = "672"
src:source.Source#456 : Source
classification = galaxy
name = "08120809-0206132"
luminosity = src:source.LuminosityMeasurement#2,
src:source.LuminosityMeasurement#553
position : SkyCoordinate
latitude = "123.033734"
longitude = " -2.103671 "
src:source.LuminosityMeasurement#2 : LuminosityMeasurement
filter = photdm-alt:PhotometryFilter#4
value = "23.2"
error = "0.04"
type = magnitude
<<modelelement>>
photdm-alt:PhotometryFilter#4 : PhotometryFilter
<<modelelement>>
spectralLocation : RealQuantity
unit = "nm"
value = "672"
src:source.LuminosityMeasurement#553 : LuminosityMeasurement
filter = phot:PhotometryFilter#32
type = magnitude
value = "23.0"
error = "0.03"
<<modelelement>>
phot:PhotometryFilter#32 : PhotometryFilter
<<modelelement>>
spectralLocation : RealQuantity
unit = "nm"
value = "672"
src:source.Source#5523 : Source
classification = star
name = "587726032792191560"
luminosity = src:source.LuminosityMeasurement#684843,
src:source.LuminosityMeasurement#472
position : SkyCoordinate
latitude = "195.49777861"
longitude = "2.50480047"
src:source.LuminosityMeasurement#472 : LuminosityMeasurement
filter = phot:PhotometryFilter#32
src:source.LuminosityMeasurement#684843 : LuminosityMeasurement
filter = photdm-alt:PhotometryFilter#4

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 composition relations, or 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 aggregation is done using a 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. The VODML elements contained in them identifies which elements are to be combined to create the object or DataType instance, what the roles are that these components play (attribute, reference).

5.3 Single-table representations and Object-Relational Mapping

Broadly speaking, this specification is all about Object-Relational Mapping (ORM). Data Models are represented in VO-DML according to an Object Oriented paradigm, although in a very limited fashion that is also suited for use in Relational Databases.

As VOTable can represent several tables in the same file with rich metadata, one can look at VOTable as a database that can represent complex relational models.

Such models are usually defined in terms of entities, with each table representing each entity, and relationships that can be expressed as tables themselves or as constraints on the values in the tables, and most often with a combination of tables and constraints. For instance, a Many-To-Many relationship between two Entities is usually represented in the relational model as a table holding IDs of instances from the tables representing the Entities, with Foreign Keys constraints.

Astronomers mainly work with single tables that hold flattened representations of relatively simple models, although in some cases complex data models are serialized in several tables inside the same file.

This specification covers both requirements. Serializations of simple models in a flattened table are easier to achieve than complex ORM mappings where information is

normalized into different tables, but they are both achievable in VOTable. Moreover, the hybrid case of partly de-normalized representations, where the model is only partly normalized, is more challenging but should also be addressable in terms of this specification. Due to the lack of a good deal of examples, it is possible that some corner cases may not be represented with this specification.
In any case, the examples in section 6, and most of section 7 (the normative part of this document) are focused on the single-table, flattened representations of instances according to some data model. Some of the patterns described in these sections are also applicable to simple ORM cases. Especially the sections dealing with mapping reference and composition relations also deal with the more complex cases of proper ORM mappings, where data is partly or completely normalized into different tables.

The simple and complex ORM patterns described by this specification usually belong to very different concrete use cases, so it should be acceptable in a broad range of cases that implementers, both on the server and on the client side, focus on the single-table mappings. Data providers requiring more complex patterns, more advanced applications, or applications built on top of standard software libraries that implement this specification as a whole will need to take advantage of the ORM mapping patterns.

6 Examples: Mapping VO-DML ⇒ VOTable

This 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 <VODML> element 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.

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 7 on how one MUST use VODML annotation in VOTable to indicate mapping to VO-DML.

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.

Notation:

- Concepts from the VO-DML metamodel are in **`Courier boldface`**
- When referring to elements form a concrete data model either by name or **`vodmlref`** we use in *italics*, e.g. *Source* or *src:source.Source*. We generally refer to data models by their short name that also should be used as prefix (but see the discussion in section 4.2). In this spec we rfer to the following models:
    - *ivoa*: TBD
    - *src*: TBD
    - *photdm-alt*: TBD

    - *vo-dml*: TBD
- VOTable concepts are normal font, generally all-caps.
- VODML/ROLE ad VODML/TYPE are used to refer to the corresponding elements of the <VODML> element.
- XML attributes have an '@' prefix

6.1 Mapping ObjectType

`ObjectTypes` consist of **`Attributes`**, **`References`** and **`Compositions`**. 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*.
We generally refer to such stand-alone objects as *Singleton Objects*. 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 refer to these as *Table Objects*. We will refer to the corresponding mapping as direct and *indirect* all through the document. This freedom of choice *where* to store objects actually 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 7.4.
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.

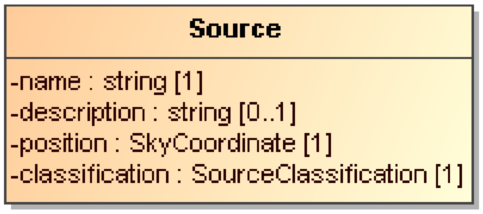

Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-classification : SourceClassification [1]

Figure 3 ObjectType representing a Source. [TBD Update to latest version of sample model]

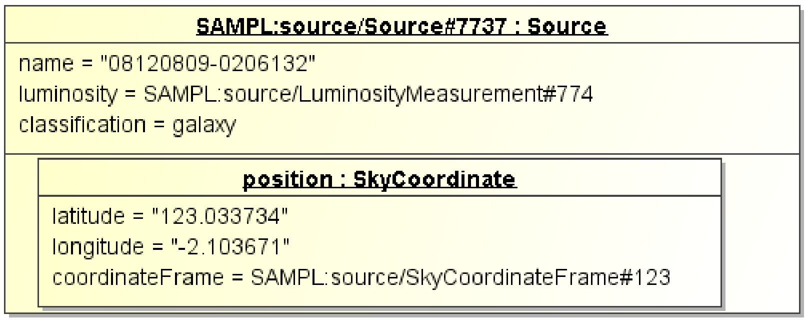

SAMPL:source/Source#7737 : Source
name = "08120809-0206132"
luminosity = SAMPL:source/LuminosityMeasurement#774
classification = galaxy
position : SkyCoordinate
latitude = "123.033734"
longitude = "-2.103671"
coordinateFrame = SAMPL:source/SkyCoordinateFrame#123

Figure 4 Instance of Source ObjectType in UML. [TBD Update to latest version of sample model]

Indirect serialization to a TABLE:
As an example of an indirect mapping (i.e. an **`ObjectType`** in a TABLE) the *Source* instance from Figure 4 is stored in the first TR in the VOTable snippet below. The TABLE is annotated using a GROUP with a VODML/TYPE indicating that it represents a *Source* from the '*src'* model. 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 *src:source,SkyCoordinate.frame* identifies a VO-DML Reference and is represented by a GROUP with a @ref attribute. When annotated with a VODML element we will refer to this pattern as a "GROUPref" (generally including the double quotes), which will be discussed in section 6.3:

```
<TABLE>
  <GROUP ID="_source">
    <VODML>TYPE>src:source.Source</TYPE></VODML>
    <FIELDref ref="_designation">
      <VODML><ROLE>vo-dml:ObjectType.ID</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_designation">
      <VODML><ROLE>src:source.Source.name</ROLE></VODML>
    </FIELDref>
    <GROUP>
      <VODML>
        <ROLE>src:source.Source.position</ROLE>
        <TYPE>src:source.SkyCoordinate</TYPE>
      </VODML>
      <FIELDref ref="_ra">
        <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
      </FIELDref>
      <FIELDref ref="_dec">
        <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
      </FIELDref>
      <GROUP ref="_icrs">
        <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
      </GROUP>
    </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, itself identified by the *vo-dml:ObjectType.ID* **`vodmlref`**. This attribute is not defined explicitly in the model, i.e. on *Source*; hence no **`vodml-id`** exists for the attribute to identify it in the *src* model

itself. In this mapping spec, which describes how to represent instances of VO-DML models in serializations, we interpret each VO-DML **`ObjectType`** as ultimately being a sub-type of *vo-dml:ObjectType* that is defined in the aforementioned *vo-dml* model. This can be compared to the way in Java all classes ultimately extend java.lang.Object, generally implicitly.
In serializations this implies we can add to any VO-DML **`ObjectType`** attributes defined on the *vo-dml:ObjectType* type. For example, since that type (which is itself a VO-DML **`ObjectType`**[11]) defines an *ID* attribute, this can be used on the representation of any **`ObjectType`**.

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 6.3).

```
<GROUP>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
  <PARAM value="08120809-0206132" ...>
    <VODML><ROLE>vo-dml:ObjectType.ID</ROLE></VODML>
  </PARAM>
  <PARAM value="08120809-0206132" ... ...>
    <VODML><ROLE>src:source.Source.name</ROLE></VODML>
  </PARAM>
  <PARAM value="galaxy" ... ...>
    <VODML><ROLE>src:source.Source.classification</ROLE></VODML>
  </PARAM>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <PARAM value="123.033734" ...>
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </PARAM>
    <PARAM value="-2.103671" ...>
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </PARAM>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </GROUP>
</GROUP>
```

Details on the mapping of the components are discussed next.

6.2 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.

[11] TBD We may want to rename this type to avoid confusion and because one can argue that it should not be a type but an instance.

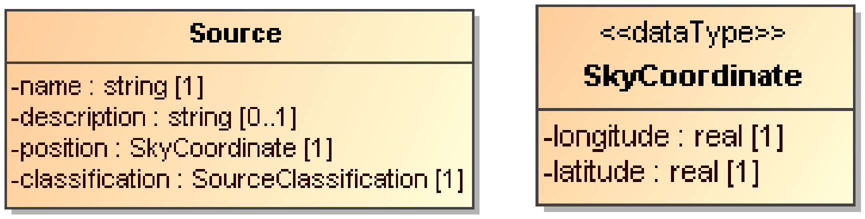

Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-classification : SourceClassification [1]
<<dataType>>
SkyCoordinate
-longitude : real [1]
-latitude : real [1]

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 the FIELD with ID=”_designation”. It does so using the **`vodmlref`** *src:source.Source.name* in the VODML/ROLE element, identifying the *name* attribute of the *Source* type. Note that in our example we use a **`vodml-id`** syntax derived from the VO-DML model itself. From this string one might 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 should inspect the formal data model to find out precisely *what* kind of model element this **`vodmlref`** represents.
Another attribute, ‘*classification*’, is represented by a PARAM as indicated by its VODML/ROLE *src:source.Source.classification* and is assigned the value ‘*galaxy*’. The attribute has as datatype an **`Enumeration`**, *SourceClassification* and indeed the PARAM defines a VALUES element with various OPTIONs (note that such a list is basically useless for a PARAM that represents a single value directly).
In this case the set of OPTIONS 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 data model. Some type of mapping is required and this is supported using the OPTION element on VODML, as explained further in section 7.5.1. The fact that this attribute is stored in a PARAM in the GROUP indicates 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. [TBD example of SDSS, with OPTION mapping]

```
<TABLE>
<GROUP>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
  <FIELDref ref="_designation" utype=" vo-dml:ObjectType.ID"/>
  <FIELDref ref="_designation" utype="src:source.Source.name"/>
  <PARAM name=”type” value=”galaxy”>
    <VODML><ROLE>src:source.Source.classification<ROLE>
      <OPTION>
        <VALUE>galaxy</VALUE>
        <LITERAL>src:source.SourceClassification.galaxy</LITERAL>
      </OPTION>
      <OPTION>
        <VALUE>star</VALUE>
        <LITERAL>src:source.SourceClassification.star</LITERAL>
      </OPTION>
    </VODML>
    <VALUES><OPTION value=”galaxy”/><OPTION value=”star”/>...</VALUES>
  </PARAM>
  <GROUP>
```

```
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <FIELDref ref="_ra">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </FILDEref>
    <FIELDref ref="_dec">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </FILDEref>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </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.

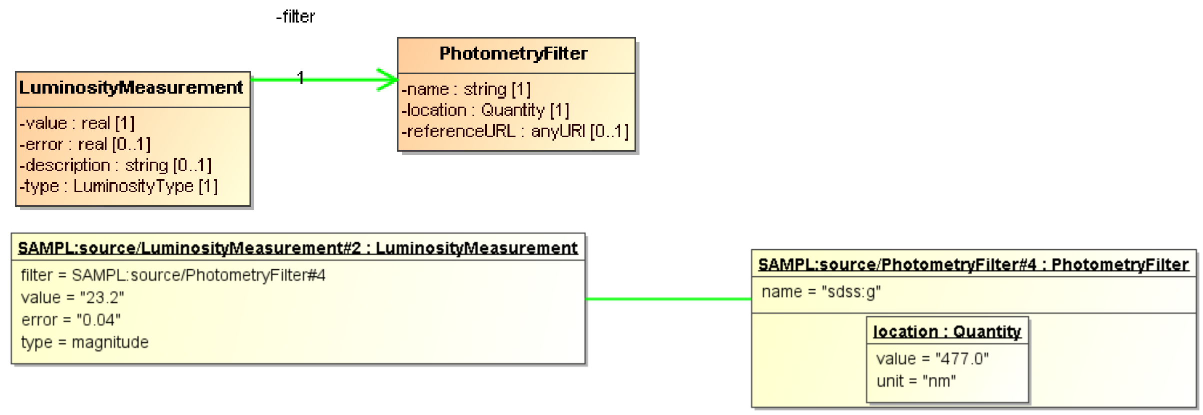
The example below shows in red a GROUP representing the attribute ‘position’ (identified by utype src:source.Source.position) that has as data type a SkyCoordinate, which itself consists of a ‘longitude’ and ‘latitude’ attribute (we defer discussing the reference to the next section). Note their structure 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>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
  <FIELDref ref="_designation">
    <VODML><ROLE>vo-dml:ObjectType.ID</ROLE></VODML>
  </FIELDref>
  <FIELDref ref="_designation">
    <VODML><ROLE>src:source.Source.name</ROLE></VODML>
  </FIELDref>
  <PARAM name=”type” value=”galaxy”>
    <VALUES><OPTION value=”galaxy”/><OPTION value=”star”/>...</VALUES>
    <VODML><ROLE>src:source.Source.classification</ROLE></VODML>
  </PARAM>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <FIELDref ref="_ra">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_dec">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </FIELDref>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </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".

6.3 Mapping Reference


-filter
1
PhotometryFilter
-name : string [1]
-location : Quantity [1]
-referenceURL : anyURI [0..1]
LuminosityMeasurement
-value : real [1]
-error : real [0..1]
-description : string [0..1]
-type : LuminosityType [1]
SAMPL:source/LuminosityMeasurement#2 : LuminosityMeasurement
filter = SAMPL:source/PhotometryFilter#4
value = "23.2"
error = "0.04"
type = magnitude
SAMPL:source/PhotometryFilter#4 : PhotometryFilter
name = "sdss:g"
location : Quantity
value = "477.0"
unit = "nm"

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.

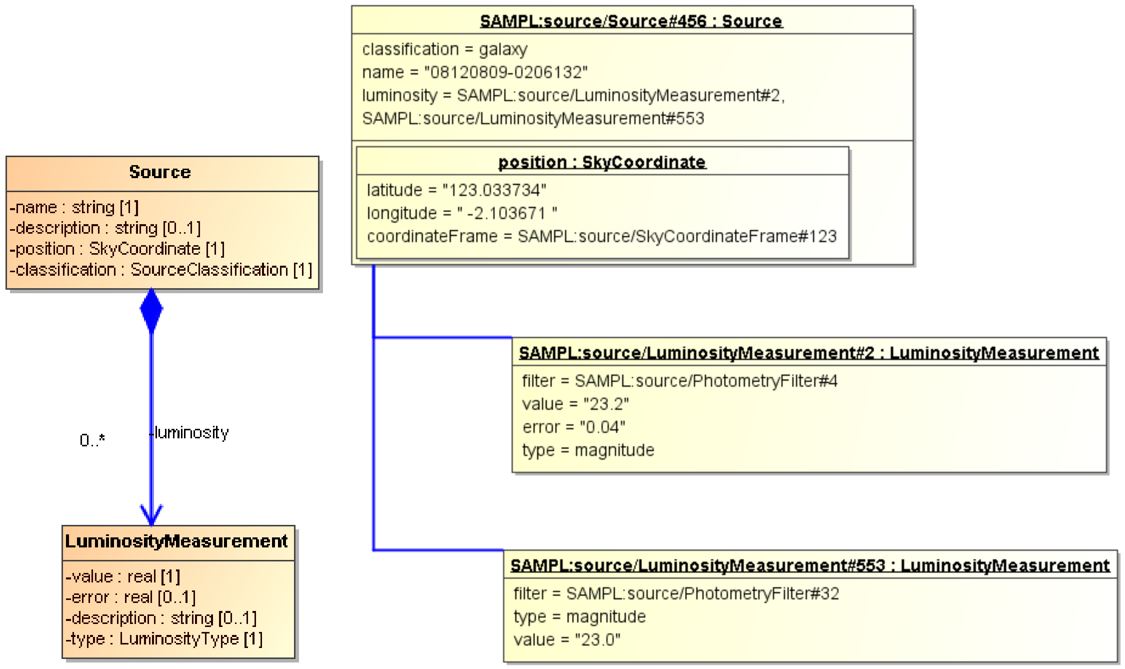
```
<GROUP ID="_icrs">
  <VODML><TYPE>src:source.SkyCoordinateFrame</TYPE></VODML>
  <PARAM value="ICRS" ...>
    <VODML><ROLE>src:source.SkyCoordinateFrame.name</ROLE></VODML>
  </PARAM>
  <PARAM value="J2000.0" ...>
    <VODML><ROLE>src:source.SkyCoordinateFrame.equinox</ROLE></VODML>
  </PARAM>
</GROUP>

<TABLE>
<GROUP>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
  <FIELDref ref="_designation">
    <VODML><ROLE>vo-dml:ObjectType.ID</ROLE></VODML>
  </FIELDref>
  <FIELDref ref="_designation">
    <VODML><ROLE>src:source.Source.name</ROLE></VODML>
  </FIELDref>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <FIELDref ref="_ra" >
```

```
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_dec">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </FIELDref>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </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>
```

6.4 Mapping Composition

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.



SAMPL:source/Source#456 : Source
classification = galaxy
name = "08120809-0206132"
luminosity = SAMPL:source/LuminosityMeasurement#2, SAMPL:source/LuminosityMeasurement#553
position : SkyCoordinate
latitude = "123.033734"
longitude = " -2.103671 "
coordinateFrame = SAMPL:source/SkyCoordinateFrame#123
Source
-name : string [1]
-description : string [0..1]
-position : SkyCoordinate [1]
-classification : SourceClassification [1]
0..*
-luminosity
LuminosityMeasurement
-value : real [1]
-error : real [0..1]
-description : string [0..1]
-type : LuminosityType [1]
SAMPL:source/LuminosityMeasurement#2 : LuminosityMeasurement
filter = SAMPL:source/PhotometryFilter#4
value = "23.2"
error = "0.04"
type = magnitude
SAMPL:source/LuminosityMeasurement#553 : LuminosityMeasurement
filter = SAMPL:source/PhotometryFilter#32
type = magnitude
value = "23.0"

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 sources 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>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
  <FIELDref ref="_designation">
    <VODML><ROLE>src:source.Source.name</ROLE></VODML>
  </FIELDref>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <FIELDref ref="_ra">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_dec">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </FIELDref>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </GROUP>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.luminosity</ROLE>
      <TYPE>src:source.LuminosityMeasurement</TYPE>
    </VODML>
    <FIELDref ref="_magJ"">
      <VODML><ROLE>src:source.LuminosityMeasurement.value</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_errJ"">
      <VODML><ROLE>src:source.LuminosityMeasurement.error</ROLE></VODML>
    </FIELDref>
    <GROUP ref="2massJ"">
      <VODML><ROLE>src:source.LuminosityMeasurement.filter</ROLE></VODML>
    </GROUP>
  </GROUP>
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.luminosity</ROLE>
      <TYPE>src:source.LuminosityMeasurement</TYPE>
    </VODML>
    <FIELDref ref="_magK">
      <VODML><ROLE>src:source.LuminosityMeasurement.value</ROLE></VODML>
    </FIELDref>
   <FIELDref ref="_errK"">
      <VODML><ROLE>src:source.LuminosityMeasurement.error</ROLE></VODML>
    </FIELDref>
    <GROUP ref="2massK"">
      <VODML><ROLE>src:source.LuminosityMeasurement.filter</ROLE></VODML>
    </GROUP>
  </GROUP>
  <GROUP">
    <VODML>
      <ROLE>src:source.Source.luminosity</ROLE>
      <TYPE>src:source.LuminosityMeasurement</TYPE>
    </VODML>
    <FIELDref ref="_magH">
      <VODML><ROLE>src:source.LuminosityMeasurement.value</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_errH"">
      <VODML><ROLE>src:source.LuminosityMeasurement.error</ROLE></VODML>
    </FIELDref>
    <GROUP ref="2massH"">
      <VODML><ROLE>src:source.LuminosityMeasurement.filter</ROLE></VODML>
    </GROUP>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" >
<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 from a collection can be directly represented inside the parent GROUP.

```
<GROUP>
   <VODML><TYPE>src:source.Source</TYPE><VODML>
   <GROUP>
     <VODML><ROLE>src:source.Source.luminosity</ROLE></VODML>
     <GROUP>
       <VODML><TYPE>src:source.LuminosityMeasurement</TYPE></VODML>
       <PARAM value="23.2" ...>
         <VODML><ROLE>src:source.LuminosityMeasurement.value</ROLE></VODML>
       </PARAM>
       <PARAM value=".04" ...>
         <VODML><ROLE>src:source.LuminosityMeasurement.error</ROLE></VODML>
       </PARAM>
       <GROUP ref="2massJ">
         <VODML><ROLE>src:source.LuminosityMeasurement.filter</ROLE></VODML>
       </GROUP>
...
</GROUP>
```

6.5 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[12]. 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 6.2, but with altered VO-DML annotation, and removal of the GROUP representing the *Source*.

```
<TABLE>
<GROUP>
  <VODML><TYPE>src:source.SkyCoordinate</TYPE></VODML>
  <FIELDref ref="_ra">
    <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
  </FIELDref>
  <FIELDref ref="_dec">
    <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
  </FIELDref>
  <GROUP ref="_icrs">
    <VODML><ROLE>src:source.SkyCoordinate.frame"</ROLE></VODML>
  </GROUP>
</GROUP>
<FIELD name="designation" ID="_designation" .../>
```

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

```
<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 ID="_icrs">
  <VODML><TYPE>src:source.SkyCoordinateFrame</TYPE></VODML>
  <PARAM value="ICRS" ...>
    <VODML><ROLE>src:source.SkyCoordinateFrame.name</ROLE></VODML>
  </PARAM>
  <PARAM value="J2000.0" ...>
    <VODML.ROLE>src:source.SkyCoordinateFrame.equinox</ROLE></VODML>
  </PAARAM>
</GROUP>
...
<GROUP name="SCS">
  <INFO value="The SCS request "/>
  <PARAM name="SR" datatype="float">
    <VODML><TYPE>ivoa:real</TYPE></VODML>
  <GROUP name="center">
    <VODML><TYPE>src:source.SkyCoordinate</TYPE></VODML>
    <INFO value="The center coordinate of the simple cone search"/>
    <PARAM name="ra" value="123.00000" datatype="float">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </PARAM>
    <PARAM name="dec" value="-2.10000"  datatype="float">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </PARAM>
    <GROUP ref="_icrs"">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </GROUP>
</GROUP>
...
```

Note, this example even assigns a VODML/TYPE to the search radius SR, identifying it as the **`PrimitiveType`** *ivoa:real*. This shows that also **`PrimitiveType`**-s and **`Enumerations`** can be used directly, i.e. outside of a role they play.

Note furthermore that the GROUP named “center” does not have a VODML/ROLE, only a VODML/TYPE, even though it is contained inside another GROUP. This is legal as a VODML annotation because that parent GROUP is *not* mapped to a data model. One could imagine the SCS protocol defining a little standard data model, say *scs* to be used in the serialization of its results. In that case also the parent group, currently named “SCS’’, could have been declared to represent, say, an *scs:SCSQuery* and *scs:SCSQuery.center* might be a possible VODML/ROLE for the child GROUP as illustrated in the following example:

```
 <GROUP ID="_icrs">
  <VODML><TYPE>src:source.SkyCoordinateFrame</TYPE></VODML>
  <PARAM value="ICRS" ...>
    <VODML><ROLE>src:source.SkyCoordinateFrame.name</ROLE></VODML>
  </PARAM>
  <PARAM value="J2000.0" ...>
    <VODML><ROLE>src:source.SkyCoordinateFrame.equinox</ROLE></VODML>
  </PARAM>
</GROUP>
...
<GROUP name="SCS">
  <VODML><TYPE>scs:SCSQuery</TYPE><VODML>
  <PARAM name="SR" datatype="float">
    <VODML><ROLE>scs:SCSQuery.SR</ROLE></VODML>
  </PARAM>
  <GROUP name="center">
    <VODML>
      <ROLE>scs:SCSQuery.center</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <PARAM name="ra" value="123.00000" datatype="float">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </PARAM>
    <PARAM name="dec" value="-2.10000"  datatype="float">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </PARAM>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </GROUP>
</GROUP>
...
```

6.6 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 name="Source">
  <VODML><TYPE>vo-dml:Model</TYPE></VODML>
```

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

In the VO-DML/XML document of the new model, the new field might have the **`vodml-id`** '*source.Source.redshift*', which translate to the **`vodmlref`** '*xsrc:source.Source.redshift*'. This **`vodmlref`** can be used to annotate instances of the extended class, like in the following example:

```
<TABLE>
<GROUP>
  <VODML>
    <TYPE>xsrc:source.Source</TYPE>
    <TYPE>src:source.Source</TYPE>
  </VODML>
  <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>
```

TBD Notice that the GROUP representing the *Source* has two VODML/TYPE elements, one referring to the subtype *xsrc:source.Source* and the other to its super type, *src:source.Source*. This possibility is a proposal on how to facilitate the work for so called *naïve clients* tailored to specific models (see Appendix B). Such clients are written only for a few specific model and only on the look-out for specific **`vodmlrefs`** from those

models. By providing all[13] types in the inheritance hierarchy above the actual type, such clients would find a type they know and they will not be confused by the unknown *xsrc* annotation.
Notice that the extending class inherits all of the parent's attributes **and their `vodmlrefs`**: this way a naïve client of the *src* model can find all the elements it may understand, while a client of the child class can **also** look for the *xsrc* elements added in the subtype definition. 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 **`vodmlrefs`** it is interested in.

7 Patterns for annotating VOTable [NORMATIVE]

In this section we list all legal mapping patterns that can be used to express how instances of VO-DML-defined types are represented in a VOTable and the possible roles they play. It defines which VOTable elements can be annotated with <VODML> elements described in section 4, what restrictions there are and how to interpret the annotation.

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, **`Composition`** (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 a mapping pattern, according to a simple "grammar" described in Appendix C. ... TBC

For example the pattern expression

```
{GROUP G| G ∊ TABLE & G/VODML/TYPE⇒ ObjectType & G/VODML/ROLE = NULL}
```

Defines the pattern: GROUP directly under a TABLE, with VODML/TYPE element identifying an **`ObjectType`** and without VODML/ROLE element.
For example [GROUP|VODML/TYPE="vo-dml:Model" ⇒ Model] implies the GROUP with VODML/TYE set to *vo-dml:Model* is mapped to a Model.
See whether we should define a list of all the legal mapping patterns in an Appendix.

Some comments on how we refer to VOTable and VO-DML elements [TBD redundant with text above in section ...]

- 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

[13] It is TBD in this proposal whether the whole type hierarchy should be replicated, or whether for example only one type is required per model.

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. @id, @ref.
- References to VO-DML elements will be capitalized and in **`courier bold`**, using their VO-DML/XSD type definitions. E.g. ObjectType, Attribute.
- Some mapping solutions require a reference to a GROUP defined elsewhere in the same VOTable. We refer to such a construct as a “GROUPref”, which is not an element of the current VOTable standard (v1.3). It refers to 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 meta-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 through its <VODML> annotation 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 an element 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.

Regarding the *normative* aspects of this specification

When we say this section is NORMATIVE we mean that:

1. when a client finds an annotation pattern conforming to one defined here, that client is justified in interpreting it as described in the comments for that pattern. It is an ANNOTATION ERROR if that were to lead to inconsistencies[14].

[14] E.g. when interpreting a GROUP as a certain ObjectType, if one of its children is not annotated or identifies a child element that is not available on the type, this is an error. Fr each pattern there is a set of rules that, if broken, are annotation errors. [TBD we better strive to make these comprehensive.]

2. when a client encounters a pattern not in this list, the client SHOULD ignore it. Interpreting it as a mapping to a data model MAY work, but is not mandated and other clients need not conform to this.

7.1 Model

7.1.1 Model declaration: GROUP in VOTABLE

Pattern expression:

```
{GROUP G| G ∊ VOTABLE & G/VODML/TYPE="vo-dml:Model"}
```

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

Restrictions

- GROUP element representing a VO-DML Model must exist directly under VOTABLE
- Each such GROUP MUST have a VODML element with VODML/TYPE="vo-dml:Model", *no* VODML/ROLE
- MUST have child PARAM element with VODML/ROLE="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 VODML/ROLE="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 VODML/ROLE="vo-dml:Model.name" and @value the name of the Model, which also works as the **`vodmlref`** prefix (see Section 4). @name is irrelevant, @datatype="char" and arraysize="*".

Example

```
<GROUP>
  <VODML><TYPE>vo-dml:Model</TYPE></VODML>
  <PARAM name="name" datatype="char" arraysize="*" value="src">
    <VODML><ROLE>vo-dml:Model.name</ROLE></VODML>
  </PARAM>
  <PARAM name="version" datatype="char" arraysize="*" value="1.0" >
    <VODML><ROLE>vo-dml:Model.version</ROLE></VODML>
  </PARAM>
  <PARAM name="url" datatype="char" arraysize="*"
      value="https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/models/sample/Source.vo-dml.xml" >
    <VODML><ROLE>vo-dml:Model.url</ROLE></VODML>
  </PARAM>
```

```
</GROUP>
```

7.2 ObjectType

See also section 7.7 on composition for more patterns regarding serialisation of `ObjectType` instances.

7.2.1 Singleton root ObjectType: direct GROUP

Pattern expression:

```
{GROUP G | G ⊂ RESOURCE & G ⊄ TABLE & G ⊄ GROUP[VODML] & G/VODML/TYPE ⇒ ObjectType
    & G/VODML/ROLE = NULL}
```

Singleton `ObjctType` represented by “direct GROUP”. Not in a TABLE, in a RESOURCE. MUST have VODML/TYPE, and no VODML/ROLE

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

7.2.2 Root ObjectType instances in Table I: not identified

Pattern expression:

```
{GROUP G | G ⊂ TABLE & G ⊄ GROUP[VODML] & G/VODML/TYPE ⇒ ObjectType
    & G/VODML/ROLE = NULL}
```

7.2.3 Root ObjectType instances in Table II: identified

Pattern expression:

```
{GROUP G, GROUP I, FIELDref F | I ∈ G & G ⊂ TABLE & G ⊄ GROUP[VODML]
    & G/VODML/TYPE ⇒ ObjectType & G/VODML/ROLE = NULL
    & I/VODML/ROLE = “vo-dml:ObjectTypeInstance.ID”
    & F ∈ I & F/VODML/ROLE = “vodml:Identifier.field”}
```

When the VOTable document stores the result of an ADQL query it may be common to find multiple copies of the same `ObjectType` instance represented in a table. [TBD example of query joining parent to child replicating parent information.] If one wants to indicate this is the case one MUST use an Identifier pattern. Here one maps one or more columns to fields in the *vo-dml:ObjectTypeInstance.ID* property that every `ObjectType` instance inherits form its implicit super-type, *vo-dml:ObjectTypeInstance*. TBC
This same identifier also allows the creation of Object-Relational Mapping patterns for references, this is described in section 7.6.3.
If such an explicitidentifier mapping does not exist, as is the case in 7.2.27.3.2, there is formally no way for clients to infer that instances in different rows are the same, even if all their properties have the same value. After all, `ObjectTypes` are *not* identified by the values of their properties, but *only* by an identifier [TBD ref to VO-DML definition].

Restrictions/conditions/rules:

- Clients SHOULD consider it an error if they encounter two objects with identical IDs, that otherwise have distinct values for the same properties.

7.3 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 7.2, 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 7.4, after the discussion of storing ObjectType instances, which is provided in section 7.2.

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.

7.3.1 Root DataType as singleton: direct GROUP

Pattern expression:

```
{GROUP G | G ⊂ RESOURCE & G ⊄ TABLE & G ⊄ GROUP[VODML]& G/VODML/TYPE ⇒ DataType
    & G/VODML/ROLE = NULL}
```

- Singleton **`DataType`** value, represented by "direct GROUP". Not in a TABLE, in a RESOURCE. MUST have VODML/TYPE, and MUST NOT have VODML/ROLE.

Restrictions:

- 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 8.2, with the PARAMs etc. providing the values of the components of the DataType.

Comments:

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 [1]). 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).

Example:
A GROUP defined as a child of RESOURCE representing a singleton datatype instance

```
<RESOURCE>
<GROUP>
  <VODML><TYPE>src:source.SkyCoordinate</TYPE></VODML>
  <INFO value="The center coordinate of the simple cone search"/>
  <PARAM name="ra" value=”123.00000” ...>
    <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
  </PARAM>
  <PARAM name="dec" value=”-2.10000” ...>
    <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
  </PARAM>
  <GROUP ref="_icrs">
    <VODML><ROLE>src:SkyCoordinate.frame</ROLE></VODML>
  </GROUP>
</GROUP>
...
```

Example:
A GROUP defined as a child of a GROUP without annotation

Note that the GROUP representing the singleton **`DataType`** is allowed to be contained in another GROUP, as long as that GROUP does not declare a VODML mapping. 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 **`Roles`** of the structured type it represents. Whether the GROUP represents the **`DataType`** instance directly or indirectly is independent of this embedding in a parent GROUP. That is purely dependent on whether the GROUP has an ancestor TABLE or not.
An example of this pattern is the case of an assumed DAL protocol, say SCS that defines *implicitly* some data structure that has components that may be represented by elements from a data model as I the following example.

Example
A GROUP representing parameters of a Simple Cone Search with components mapped to data model elements.

```
<RESOURCE>

<GROUP>
 <INFO value="The simple cone search request"/>
 <PARAM name=”serchRadius” value=”3” unit=”arcsec” .../>
 <GROUP name=”center”>
  <VODML><TYPE>src:source.SkyCoordinate</TYPE></VODML>
  <INFO value="The center coordinate of the simple cone search"/>
  <PARAM name="ra" value=”123.00000” ...>
```

```
    <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
  </PARAM>
  <PARAM name="dec" value="-2.10000" ...>
    <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
  </PARAM>
  <GROUP ref="_icrs">
    <VODML><ROLE>src:SkyCoordinate.frame</ROLE></VODML>
  </GROUP>
 </GROUP>
...
```

7.3.2 Root DataType values in TABLE records: indirect GROUP

Pattern expression:

```
{GROUP G | G ⊂ TABLE & G ⊄ GROUP[VODML] & G/VODML/TYPE ⇒ DataType  & G/VODML/ROLE = NULL}
```

Restrictions:

- ...

Comments:

A GROUP defined on a TABLE, annotated with a **DataType**, represents a **DataType** instance indirectly.

Example

```
<TABLE>
  <GROUP>
    <VODML><TYPE>src:source.SkyCoordinate</TYPE></VODML>
    <FIELDref ref="_ra">
      <VODML><ROLE>bSTC:SkyCoordinate.longitude</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_dec">
      <VODML><ROLE>bSTC:SkyCoordinate.latitude</ROLE></VODML>
    </FIELDref>
    <GROUP ref="_icrs">
      <VODML><ROLE>bSTC:SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </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>
...
```

7.3.3 DataType as Attribute: GROUP in GROUP

Pattern Expression:

```
{GROUP A, GROUP G | A ∈ G & A/VODML/ROLE ⇒ Attribute & G/VODML ⇒ StructuredType}
```

- A GROUP representing a structured type can have Attributes that are of a structured type as well.
- However, this time the nested GROUP has a VODML/ROLE identifying the Attribute that the GROUP represents.

Restrictions:

- The **Attribute** identified by GROUP A's VODML/ROLE MUST be available to the structured type represented by the containing GROUP (G).
- GROUP A SHOULD also declare the actual structured type it represents, through its VODML/TYPE. This type MUST be a valid type for the **Attribute**, i.e. the **Attribute's** declared type or one of its subtypes. If the type in the serialization is a subtype, the VODML/TYPE MUST declare that.

Comments:

Example

```
<GROUP>
  <VODML><TYPE>src:source.Source</TYPE></VODML>
...
  <GROUP>
    <VODML>
      <ROLE>src:source.Source.position</ROLE>
      <TYPE>src:source.SkyCoordinate</TYPE>
    </VODML>
    <FIELDref ref="_ra">
      <VODML><ROLE>src:source.SkyCoordinate.longitude</ROLE></VODML>
    </FIELDref>
    <FIELDref ref="_dec">
      <VODML><ROLE>src:source.SkyCoordinate.latitude</ROLE></VODML>
    </FIELDref>
    <GROUP ref="_icrs">
      <VODML><ROLE>src:source.SkyCoordinate.frame</ROLE></VODML>
    </GROUP>
  </GROUP>
...
</GROUP>
```

7.4 PrimitiveType and Enumeration

PrimitiveTypes and **Enumerations** represent atomic values. They appear in serializations of data models predominantly in their role as attributes defined on structured types (**ObjectType** or **DataType**) and this specification focuses on that role. There is only limited extra information in identifying an individual FIELD or PARAM as representing some primitive type or enumeration from a data model, without indicating the role the FIELD or PARAM plays in some complex type. VO-DML assumes the existence of a standard data model (*ivoa,* see [1] TBD refer to actual model) defining primitive types such as *ivoa:integer*, *ivoa:string* etc. But there is no new semantics implied by these types beyond the ones their counterparts in the VOTable schema carry[15].

7.4.1 Singleton primitive value as PARAM

Pattern expression:

```
{PARAM P| P ∉ G[VODML] & P/VODML/ROLE = NULL & G/VODML/TYPE ⇒ PrimitiveType|Enumeration}
```

This indicates a PARAM that is *not* contained in a VODML-annotated GROUP, has a VODML/TYPE identifying an atomic type and no VODML/ROLE.
TBD do we want to support this? For if so, what about FIELDs?

[15] TBD how about ivoa types that do not exist in VOTable such as *ivoa:datetime*?

7.4.2 `EnumLiteral` as OPTION

Pattern expression:

```
{FIELDref FR, FIELD F, OPTION O| FR/@ref=F/@ID & O ⊂ F
   & FR/VODML ⇒ Enumeration & FR/VODML/OPTION/NAME = O/@value
                             & FR/VODML/OPTION/LITERAL ⇒ EnumLiteral}
```

TBC

- VODML/OPTION
- value to literal
- value to SKOS concept

7.4.3 `SKOSConcept` as OPTION

TBD

7.5 Attributes

7.5.1 Attribute values in FIELD: FIELDref in GROUP in TABLE

Pattern expression:

```
{FIELDref F, GROUP G| F ∊ G & G ⊂ TABLE & F/VODML/ROLE ⇒ Attribute
     & G/VODML ⇒ StructuredType }
```

- A FIELDref contained in a GROUP represents an Attribute serialized by the FIELD it refers to.

Restrictions

- The **`Attribute`** identified by the FIELDref/VODML/ROLE MUST be available to the structured type represented by the containing GROUP (G).
- FIELDref MUST reference a FIELD in the TABLE that contains its parent GROUP.
- The **`Attribute`** MUST have an atomic datatype, or a **`DataType`** from the set of special cases described in be section 7.9.
- The @datatype of the referenced FIELD SHOULD be compatible with the declared **`datatype`** of the **`Attribute`**. E.g. an integer (*ivoa:integer*) **`Attribute`** SHOULD be represented by VOTable's int or long.

Example

See example in 6.2.

7.5.2 Attribute to PARAM in GROUP

Pattern Expression:

```
{PARAM P, GROUP G | P ∊G & G/VODML/ROLE ⇒Attribute & G/VODML ⇒ StructuredType}
```

- 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`**'s **`datatype`** MUST identify an atomic type, i.e. a **`PrimitiveType`** or **`Enumeration`**, or a **`DataType`** from the set of special cases described in be section 7.9.
- The PARAM @datatype SHOULD be a compatible, valid serialization type for the **`datatype`** of the **`Attribute`**.

Example

See example in TBD

7.5.3 Attribute to PARAMref in GROUP

Pattern expression:

```
{PARAMref P, GROUP G | P ∊ G & P/VODML/ROLE ⇒ Attribute & GROUP/VODML ⇒ StructuredType]
```

Inside a GROUP a PARAMref identifies a PARAM defined inside the same RESOURCE or TABLE where the GROUP is defined. Using its VODML/ROLEthe 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 G.
- The PARAM referencd by the PARAMref MUST obey the restrictions defined for the annotation of a PARAM by the **`Attribute's`** type in section 7.5.2.

Example

See example in 6.2

7.5.4 DataType as Attribute: "GROUPref" in GROUP

A PARAMref identifyin some PARAMin a GROUP can represent an **`Attribute`** with an atomic type on a **`StructuredType`**, so we can to a GROUP representing a structured **`DataType`** to represent **`Attributes`**.

7.6 References

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 defined on the referrer; it is one of the roles an **`ObjectType`** can play in the definition of another structured type. **`Reference`** is a many-to-1 relation, many referrers can reference the same target object. The implication of a Reference relation is that the referenced object is somehow "used" by the referrer. Often, especially when the referrer is a **`DataType`**, it defines reference data that helps interpret the values its attributes assume. See the VO-DML document [1] for more information.

In explicit serialization languages [TBD must define and discuss this term earlier in this document, say in section 3] such as XML Schema or the relational model there is support for representing this relation. E.g. XML schema allows one to define an attribute of type ID, which must have a value unique in the document. Another element can "refer" to this element using an attribute of type IDREF. It does so by assigning a value to the IDERF attribute and that same value MUST have been assigned to the ID attribute of another element for the document to be valid. This equality of values implies the relationship between the two elements, but there is in the definition no restriction on the type of the referenced element.

If a restriction on the referenced element is desired, XML Schema also allows one to define a "key/keyref" combination. A *key* has a unique name and is defined using an XPath [REF] "selector", identifying a set of elements in a document and one or more fields that define the unique values distinguishing between these elements. A *keyref* can similarly be defined by the name of the target key, a selector identifying the referrer elements and the fields used for identifying the specific referenced elements among the ones defined by the key.

In a relational database a column on one table can be defined as a (primary) key, its values must be unique in that table[16]. Another table (or even the same one) can declare a column to be a "foreign key", which "references" the first table through its key column.

In all three cases the reference relationship is implemented using pairs of elements which take up the same value, the ID or key, being referenced using the IDREF and keyref/foreign key column(s). These elements can be seen as addresses and pointers, the way by which similar relationships can be made in programming languages, sometimes explicit, as in C/C++, sometimes implicit as in Java/C#.

The mapping prescription laid out in this specification follows the same approach. It identifies three different patterns for representing references between instances of **`Types`**. Their difference lies in the way the target object is represented and identified, which puts demands on the way to identify them on the referrer objects. If the target object is serialized in the same document as the referrer it may be either represented as a singleton object represented by a direct Group (7.2.1) or as an object in a table (7.2.2 – 7.2.3).

The specification also allows for the case where the target object is *not* represented in the same VOTable document as the referrer. It may exist in another VOTable, or in some alternative, possibly standardized serialized form. An example of the latter could be an XML document following a standardized XML schema explicitly defined to represent the data model, or possibly a location in some publicly accessible database with a relational schema mapped to the model in a standardized Object-Relational manner.

This "remote reference pattern" is potentially very useful. By locating certain standard, reference data objects in some well-defined place, possibly a registry, different documents can reference the same instance in an explicit and interoperable manner. In this case, instead of providing a serialization of (part of) the referenced object with every file that uses it, a reference to the external serialization could be sufficient.

As an example consider defining a set of standard space-time coordinate reference frames and registering these in an IVOA sanctioned registry, with a well-defined and

[16] One can use multiple columns, that together must be unique (and NOT NULL!).

unique IVOA Identifier. Client tools could be coded to understand references to such standard data sets, so that their identifier might suffice to express the reference. Alternatively they might cache such globally persistent instances so to minimize de-referencing. Proper support for such standard reference data requires some standardized form for their registration and storage, an effort that should be part of the creation of standard data models and should have maintenance responsibilities similar to that of the UCD vocabulary [8].
In all patterns below the actual **Reference** is represented by a GROUP contained by the GROUP representing the referrer, and with structure that allows the identification of the target object. We do this for uniformity: there may be cases where a single reference PARAM for example might be sufficient, just as an IDREF can identify a target with an ID in XML Schema, but more generic cases require more structure. And in fact our solution for this simplest case, using what we will refer to as a "GROUPref" (including the quotes), is more lightweight even that a solution with a PARAM as will be shown next.

7.6.1 Reference to singleton Object in direct GROUP

Pattern Expression:

```
{GROUP S, GROUP R, GROUP T|
    S/VODML ⇒ StructuredType & R ∈ S & R/@ref = T@ID
    & T/VODML ⇒ ObjectType & R/VODML/ROLE ⇒ Reference
    & T ∉ TABLE & T ⊂ RESOURCE
    [& R/VODML/TYPE = "vo-dml:GROUPref"]
}
```

This pattern represents a source GROUP R (the *reference*), contained in a GROUP S (the *referrer*) representing a **StructuredType**, referencing a *target* GROUP T, representing a singleton **ObjectType** instance. The reference R uses its @ref attribute to identify the target GROUP T which is identified by a unique @ID. The GROUP R has no elements apart from the VODML element which MUST contain a ROLE identifying a particular **reference**; it MAY explicitly identify its VODML/TYPE as being *vo-dml:GROUPref*.
We refer to this pattern represented by GROUP R as a "GROUPref", it is a reference to an identified GROUP. But we use the quotes as we do not need an explicit GROUPref element in the sense of VOTABLE's FIELDref and PARAMref to represent this concept. It can also be used in other patterns as shown in the next section.

Restrictions

- The **Reference** identified by R/VODML/ROLE MUST be *available* on the **StructuredType** identified by O/VODML.
- **OjectType** identified by T/VODML MUST be compatible with the **datatype** of the **Reference**, i.e. must either be the same type or one of its subtypes.
- The GROUP R representing the **Reference** must not have any child elements apart from the VODML element.
-

Example

```
<GROUP ID="_2massJ">
  <VODML><TYPE>phot:PhotometryFilter</TYPE></VODML>
  <PARAM name="name" datatype="char" value="2mass:J">
```

```
    <VODML><ROLE>phot:PhotometryFilter.name</ROLE></VODML>
  </PARAM>
...
</GROUP>
...
<GROUP>
  <VODML><TYPE>src:source/Source</TYPE></VODML>
  <GROUP>
    <VODML>
      <ROLE>src:source/Source.luminosity</ROLE>
      <TYPE>src:source/LuminosityMeasurement</TYPE>
    </VODML>
    <FIELDref ref="_magJ" utype="SAMPL:source/LuminosityMeasurement.value"/>
    <FIELDref ref="_errJ" utype="SAMPL:source/LuminosityMeasurement.error"/>
    <GROUP  ref="_2massJ">
      <VODML><ROLE>src:source/LuminosityMeasurement.filter</ROLE></VODML>
  </GROUP>
...
```

7.6.2 Reference to object(s) in a TABLE I: objects in same row

Pattern Expression:

```
{GROUP S, GROUP R, GROUP T, TABLE TB|
    S/VODML ⇒ StructuredType & R ∊ S & R/@ref = T@ID
    & TB/VODML ⇒ ObjectType & R/VODML/ROLE ⇒ Reference
    & S ⊂ TB & T ⊂ TB
    [& R/VODML/TYPE = "vo-dml:GROUPref"]
}
```

This pattern represents a "reference GROUP" R, contained in "source GROUP" S representing a structured type, referencing a "target GROUP T". Both S and T are contained in the same TABLE TB and a "GROUPref" is used to represent the reference. This pattern indicates that each row in the TABLE TB contains (at least) two objects, represented by GROUPS S and T, and that S references T through the reference represented by GROUP R. R has no elements apart from the VODML element; it MUST identify the actual **reference** in its VODML/ROLE element; it SHOULD explicitly identify its VODML/TYPE as being v*o-dml:GROUPref*; it MUST identify which reference it represents through its VODML/ROLE.

Example:
TBD

7.6.3 Reference to object(s) in a TABLE II: objects in different rows, possibly different TABLE

Pattern Expression:

```
{GROUP S, GROUP R, GROUP T, GROUP PK, GROUP FK |
    S ⊂ TABLE & T ⊂ TABLE
    & S/VODML ⇒ StructuredType
    & T/VODML ⇒ ObjectType
    & PK ∊ T & PK/VODML/ROLE="vo-dml:ObjectTypeInstance.ID"
    & R ∊ S & R/VODML/ROLE ⇒ reference & R/VODML/TYPE = "vo-dml:ORMReference"
    & FK ∊ R & FK/VODML/ROLE="vo-dml:ObjectTypeInstance.ID"
    [& R/@ref = T@ID]
}
```

This is the most complex pattern in this specification. It is the analogue of a foreign key constraint in a relational database. The pattern represents a “reference GROUP” R, contained in “source GROUP” S representing a structured type, referencing a “target GROUP T”. S and T may be contained in the same TABLE, but in general this need not be the case.

This pattern indicates that each row in the TABLE annotated by GROUP S contains a corresponding structured object, which has a reference to an object represented by GROUP T, located in some row in its TABLE. The reference is represented by the GROUP R. The GROUP R MUST identify which reference it represents through its VODML/ROLE. R MUST explicitly identify its VODML/TYPE as being *vo-dml:ORMReference*.

This latter requirement indicates that this pattern represents an object-relational mapping pattern. It is clear that in contrast to the previous two cases the combination of S/@ref and T/@ID is not sufficient to identify which objects are related. To make the connection this specification prescribes that the relation must be based on a kind of foreign key pattern.

First, the objects T MUST be explicitly identified as described in 7.2.3. So T MUST contain a GROUP PK with VODML/ROLE=”*vo-dml:ObjectTypeInstance.ID*” and I must contain components which allow one to distinguish between different instances of type T in table T2. This may be a unique[17] identifier, but this is not mandatory.

To identify which T is referenced, the reference GROUP R MUST contain a “foreign key GROUP” FK that also has VODML/ROLE=”*vo-dml:ObjectTypeInstance.ID*”. It’s structure MUST be compatible with the structure of the GROUP PK. In general this means that if PK contains *N* FIELDrefs, also FK must contain *N* FIELDrefs. In general N=1, but if not, this specification prescribes that the order is important. I.e. FK’s FIELDref 1 must be matched to PKs FIELDref 1 etc. to find the referenced objects. Note also that it is allowed that FIELDref-s in PK are matched by PARAMs in FK. [TBD other way as well?]

Example:

TBD

7.6.4 Reference to Object in external data store

Pattern Expression:

```
{GROUP O, GROUP S, PARAM P|
    O/VODML ⇒ StructuredType & S ∊O & S/@ref = NULL
    & S/VODML/ROLE ⇒ Reference
    & S/VODML/TYPE = “vo-dml:RemoteReference”
    & P ∊ S & (P/VODML/ROLE = “vo-dml:RemoteReference.ivoid” |
               P/VODML/ROLE = “vo-dml:RemoteReference.uri”)
}
```

This pattern identifies a reference to an object serialized elsewhere. Such an instance MUST be identified using an IVOA Identifier [12] if it is stored in an IVOA Registry. However, a URL might be provided as a convenient hook for the client. The reference is again represented by a GROUP, but now with PARAM whose @value is an IVOA Identifier. As alternative one can use a URI, but exactly *how* that should be resolved is outside of the scope of this document.

Example:

[17] As discussed earlier, it is allowed that the same object is represented multiple times in the same table.

For example the previous example from the previous sub-section could be written as follows:

```
<GROUP>
  <VODML><TYPE>src:source/Source</TYPE></VODML>
  <GROUP>
    <VODML>
      <ROLE>src:source/Source.luminosity</ROLE>
      <TYPE>src:source/LuminosityMeasurement</TYPE>
    </VODML>
   <FIELDref ref="_magJ" utype="SAMPL:source/LuminosityMeasurement.value"/>
   <FIELDref ref="_errJ" utype="SAMPL:source/LuminosityMeasurement.error"/>
   <GROUP  ref="_2massJ">
     <VODML>
       <ROLE>src:source/LuminosityMeasurement.filter</ROLE>
       <TYPE>vo-dml:RemoteReference</TYPE>
     </VODML>
     <PARAM name="ivoid" datatype="char" arraysize="*"
            value="ivo://ivoa.registry/dm/phot/instances/2mass.J">
       <VODML><ROLE>vo-dml:RemoteReference.ivoid</ROLE></VODML>
     </PARAM>
     <PARAM name="ivoid" datatype="char" arraysize="*"
            value="http://ivoa.registry/dm/phot/instances/2mass#J">
       <VODML><ROLE>vo-dml:RemoteReference.uri</ROLE></VODML>
     </PARAM>
  </GROUP>
...
```

Here it is assumed that the DM working group maintains a set of data model instance documents for various instruments and stores these in some registry. [TBD check how realistic the ivoId is, add URL.]

7.7 Composition

A **Composition** is defined in VO-DML as a "whole-parts" relation between parent and child **ObjectTypes**. The child can be interpreted as defining a *part* of the parent, the parent is "composed of" the child objects. Generally a parent can have a collection of 0..N children of a certain type, where N may be unbounded, but the multiplicity can be more constrained. The relation between the child and its parent is much stronger than the relation between a source and a target in a **Reference** relation. In particular the child's life-cycle is tightly coupled to its parent: a child only has one parent and when when the parent is deleted, so is the child. Because of this also the serialization of the relationship may in many cases be more direct than the indirect reference mapping patterns of the previous section.
For example, in faithful XML serializations of a data model one will generally map child objects as elements directly contained by the parent element. It is possible to do so in VOTable as well, having a child object represented by a GROUP that is contained within the GROUP representing the parent. This pattern is the most natural for this relationship because a collection element is really to be considered as a *part* of the parent object.
This containment of GROUPs can be used for singleton patterns in an obvious manner. But it may also be used to represent a flattening of the parent-child relation in a TABLE. 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 from a collection of photometry points contained by the sources. (See the sample data model).
Faithful object-relational mapping of composition relations does generally not allow for such a containment. The typical pattern is that the table representing the child

ObjectType has a foreign key to the table representing the parent. This specification allows for this mapping pattern as well, using the ORMReference pattern from section 7.6.3 in combination with the implicit *vo-dml:ObjectType.container* **Reference**.

7.7.1 Composition I: child GROUP inside parent GROUP

Pattern Expression:

```
{GROUP P, GROUP C | C ∊P & P/VODML ⇒ ObjectType & C/VODML/ROLE ⇒Composition
    [& C/VODML/TYPE ⇒ObjectType]}
```

GROUP C represents a child object in a **collection** on some **ObjectTupe** identified by its VODML/ROLE, and of **datatype** possibly identified by its VODML/TYPE. In this pattern C is contained inside of a GROUP P that represents an **ObjectType** that is a valid **Container** of the **Collection**.
No statement is made about the location of the parent GROUP. If it is inside a RESOURCE, *not* in a TABLE, both parent and child represent singleton objects. If it is inside a TABLE, each row stores both parent and child objects.
Note also that there may be multiple child GROUPs identifying the same composition relation in the same parent GROUP.

Example
See example in 6.4

7.7.2 Composition II: object-relational reference from child to parent, both in TABLEs

Every instance of an **ObjectType** inherits from its implicit ultimate super type *vo-dml:ObjectTypeInstance* its *vo-dml:ObjectTypeInstance.container* property. In the *vo-dml* model this is represented as a collection of type *vo-dml:Reference*, with multiplicity 0..1. This should be interpreted as the possibility of adding on a serialized **ObjectType** instance a reference to its container/parent object.
In the case of serializing to a VOTable, this allows one to implement an object relational mapping pattern for the parent-child relation as described in the introduction to this section.

Pattern Expression:

```
{GROUP CH, GROUP R, GROUP P, GROUP PK, GROUP FK |
    CH ⊂ TABLE & P ⊂ TABLE
    & CH/VODML ⇒ ObjectType
    & P/VODML ⇒ ObjectType
    & PK ∊ P & PK/VODML/ROLE=”vo-dml:ObjectTypeInstance.ID”
    & R ∊ CH & R/VODML/ROLE = “vo-dml:ObjectTypeInstance.container”
            & R/VODML/TYPE = “vo-dml:ORMReference”
    & FK ∊ R & FK/VODML/ROLE=”vo-dml:ObjectTypeInstance.ID”
    [& CH/@ref = P@ID]
}
```

The GROUP CH represents a collection of child objects that are contained in parent objects represented by GROUP P. CH and P are generally stored in different tables. CH references P following the ORM reference pattern described in section 7.6.3 above. The only constraint is that the VODML/ROLE on the GROUP representing the reference MUST be *vo-dml:ObjectTypeInstance.container*.

For clients it is useful if an identification is made *which* collection on the parent object the type represented by GROUP CH belongs to. The VO-DML spec restricts types to be the child in at most one composition relation, where inheritance/polymorphism is taken into account[18]. Hence that information is in principle available through the VO-DML model and an explicit indication would be redundant, but it may make implementation of clients simpler.
One way to make this reference is to allow the GROUP CH to have a VODML/ROLE identifying the collection as well as the VODML/TYPE. An objection to this could be if we want to keep the invariant that only those GROUPs can have a VODML/ROLE that are contained inside another GROUP defining a type on which the role is available [TBD ref to statement along these lines in this spec, *if* we impose this rule]. The following pattern is a possible implementation that preserves this invariant:

```
{GROUP CH, GROUP R, PARAM P |
    P ∊ R & P/VODML/ROLE = "vo-dml:ORReference.container"
          & P/@value ⇒ collection
          & P/@datatype = "string"
          [ & P/VODML/TYPE = "vo-dml:ref"]
}
```

The PARAM on the GROUP R identifies the collection, which MUST be available on the **ObjectType** represented by the GROUP P.

Annotators SHOULD add a reference from the parent group to the group representing the collection of child objects according to the following pattern:

```
{GROUP P, GROUP C, GROUP CH|
      C ∊ P & C/VODML/ROLE = collection
            & C/VODML/TYPE = "vo-dml:CollectionReference"
    & C/@ref = CH@ID
}
```

It indicates that the GROUP representing the container P contains a GROUP C representing a "collection reference" to the GROUP with the child objects C. It does *not* indicate that all objects in C are contained in P. And the child objects MUST have the container reference as defined in the first pattern above.

TBD Do we want these extra patterns? They imply a redundancy with the first. Thatone is sufficient to define the relation and necessary as it defines how to identify the parent on the child. The 3rd pattern does allow parsers to start looking for the children as soon as the collection-reference is encountered.
TBD should we merge these patterns into one? Would make the pattern even more complex.

Example:
TBD

7.8 Extends, inheritance

The inheritance relation between two **Type**-s might seem not to be relevant for serializations and mapping. After all the relation implies no relation between objects, the

[18] I.e. if a super-type is declared as the child in a composition relation, this "property" is inherited by its sub-types.

target of serializations, only between types. In serializations, any inherited **`Role`** can appear on a sub-type and, as described above, all properties inherited from a super-type are identified with the *vodml-id* defined on the super-type.
There are some special cases though where inheritance makes a non-trivial appearance. The first we have seen already, in that we can explicitly *cast* a **`Role's`** value to a sub-type of a declared datatype. Supporting this type of polymorphism at the serialization level is one main motivation for adding <TYPE> to the <VODML> element. See Appendix B for a discussion how different types of clients can deal with this.
There are however subtler issues related to inheritance mapping. An example comes from a typical object-relational mapping pattern for inheritance hierarchies. When creating a relational model for an object model with inheritance hierarchies, usually there are three patterns people will follow.
One may choose to map each concrete (i.e. non-abstract) type to its own table. But one may also choose to map all sup-types of a given (ultimate) super type to a single table. Different rows may store different sub-types, and certain columns defined for storing an attribute defined on one sub-type will in general be NULL on its siblings. Or one may choose a mixture of these where there is a table for each type, but these only store elements defined on that type; sub-types have a foreign key to their parent and to retrieve a complete subtype this pointer must be followed, possibly recursively until the ultimate super-type. Tools such as Hibernate [7] provide explicit support for these inheritance mapping patterns, and we point the reader to their documentation for more information.
In the latter two patterns there will generally be a table, either the single one storing the whole hierarchy, or the one representing the ultimate super-type, which defines a special column that allows one to decide which concrete sub-type is stored in a particular row. This may be through some pre-agreed code, or by storing the complete name of the type. This pattern can be supported as well.

Pattern Expression:

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).

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.

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 above, where the child representation carries over the vo-dml:Instance.type from its ancestors only applies to inter-Model extensions, UNLESS the extended type is abstract. In this case, clients may want to look for the abstract type before interpreting the actual concrete type. For instance, a client may look for all the "*src:source.SkyError*" instances before interpreting them as a circle error, an elliple error, and so on.

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.

7.9 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.

8 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.

8.1 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.

8.2 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.

8.3 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.

8.4 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 or RESOURCE*. 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 again enables backward compatibility.

9 Serializing to other file formats

VOTable is expressive enough to allow the mapping patterns described in this specification. Other formats (notably FITS) cannot support such annotations. In particular FITS does not support the GROUP-ing of columns, which is of prime importance for the current specification. A solution to this, which is even one of the motivations behind the VOTable specification, is to annotate other format's tables with a VOTable *wrapper*. VOTable (see [2], section 5.2) already supports wrapping FITS tables using a TABLE/DATA/FITS. This has a STREAM/@href URL to specify which FITS file is represented and an @extnum attribute to indicate which FITS extension is used.
Some FITS formats also allow a VOTable header to be included in an HDU. Whilst this would allow data and meta-data to be combined in a single file, this solution seems to be less portable, since some FITS readers do not support these files.
A similar solution could be used to annotate the tables in the schema of a TAP service. Here no specific support yet exists, but could be easily added to the TAP specification. I

fact a VOTable representing all TAP tables as "empty" TABLE elements with only header, no DATA information has been proposed in the past as one of the ways by which a TAP schema's metadata could be declared. If this were to be a formalized option it would automatically allow the mapping patterns from the current specification to be included. The main challenge for TAP is whether that annotation can be carried over to the results of queries against its schema. For the simple, column-only annotations such as UCDs or descriptions, this is already a non-trivial task, and not always possible[19]. General ADQL queries can easily unravel the GROUP-ing of columns and their nesting and produce results that cannot be annotated with concepts of the original data models. However we believe this is a 90-10 problem, and for the most common queries it should be possible to identify the data model concepts with not much more work than is already required to carry along the UCDs.
This should in particular not be a problem for the results of standard protocols such as the simple cone search. There the service provider is in charge of designing the result set, which, being tabular, can be easily annotated using the current specification. In fact, in the absence of a TAP_SCHEMA-like mechanism, it could be argued that the current specification presents a perfect mechanism by which an SCS service can declare the contents of its holdings.
Whereas similar approaches can likely be used for other tabular formats, an interesting question is whether mapping patterns can be defined also for annotating serialization formats other than tables. In particular the more structured formats such as XML or JSON, or more modern ones such as Google Protocol Buffers[20] or Apache Avro[21], would pose different problems.
The first approach would be to see if and how such formats can be used to provide *faithful* representations as defined in section 3. For XML it has been shown in a similar effort (see [3] and [13]) that one can derive an XML schema that represents the types and relations of a well-defined data model in a 1-1 fashion. It is even possible to do so in an automated manner using code generation and we believe it would be a useful effort to define such . As long as a well-defined meta-model exists for the target serialization, and as long as it is rich enough we believe this will carry over to other serialization formats.
The problem is when one want to annotate existing XML documents that have not been designed as such. It has as yet not been explore whether a simple annotation mechanism such as the current one will work there as well. We believe that *if* the concepts of a data model can be identified in an informal manner, it may be possible to at least partially reproduce the target serialization as a *view* on a faithful serialization. For example one might define an XSLT document that, when working on a faithfully serialized data model would produce the desired one.
More useful would likely be the opposite, i.e. a document that would produce a faithful serialization of a non-faithful one. Such a document would allow clients to first transform the document to a known serialization for further processing. This brings one close to the global-as-view approach, in that the global schema, the data model, is represented as a view on the source.

More discussion will be needed to address these issues.

[19] E.g. consider `SELECT a-b FROM foo`. Even if columns `a` and `b` have a UCD, what is the UCD of their difference?
[20] https://developers.google.com/protocol-buffers/
[21] https://avro.apache.org/

10 References

[1]. VO-DML document

[2]. VOTable document(s)

[3]. Simulation DM spec

[4]. Characterization spec

[5]. Spectrum DM spec

[6]. STC spec

[7]. ORM mapping tools: Hibernate, JPA spec

[8]. UCDs

[9]. Investigation into existing UTYPE usages, document form UTYPEs tiger team.

[10]. Note on STC in VOTable (M. Dempleitner)

[11]. PR on use of SKOS vocabularies in IVOA

[12]. IVOA Identifiers

[13]. VO-URP

Appendix A. The VODML annotation elements in the VOTable schema

The VOTable's *Param*, *Paramref*, *Fieldref* and *Group* definitions obtain an extra element named <VODML> with type *VODMLAnnotation*. That type is defined by the next fragment extracted from the proposal VOTable-1.3_vodml.xsd (as of volute 2873).

```
<!-- VODMLAnnotation section -->

  <xs:simpleType name="VODMLReference">
    <xs:annotation>
      <xs:documentation>
      The valid format of a reference to a VO-DML element. (Used to be 'UTYPE').
      MUST have a prefix that elsewhere in the VOTable is defined to correspond to a VO-
DML model defining the referenced element.
      See "mapping document", https://volute.googlecode.com/svn/trunk/projects/dm/vo-
dml/doc/UTYPEs-WD-v1.0.pdf.
      Suffix, separated from the prefix by a ':', MUST correspond to the vodml-id of the
referenced element in the VO-DML/XML representation
      of that model.
      </xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:pattern value="[\w_-]+:[\w_\-/\./*]+" />
    </xs:restriction>
  </xs:simpleType>

 <xs:complexType name="VODMLOptionMapping">
 <xs:annotation>
   <xs:documentation>
   Allows one to map particular values defined in a VALUES/OPTION list to enumerayion
literals
   in the VO-DML model.
   </xs:documentation>
 </xs:annotation>
  <xs:sequence>
```

```
    <xs:element name="OPTION" type="xs:string" minOccurs="1" maxOccurs="1" />
    <xs:element name="LITERAL" type="VODMLReference" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
 </xs:complexType>
<!--
-->
 <xs:complexType name="VODMLAnnotation">
  <xs:sequence>
    <xs:element name="ROLE" type="VODMLReference" minOccurs="0">
     <xs:annotation>
      <xs:documentation>
       MUST be provided on annotations of FIELDref and PARAMref and, if contained in a
VO-DML annotated GROUP or PARAM.
       If not provided on a root GROUP, indicates that it represents a stand-alone
object.
      </xs:documentation>
     </xs:annotation>
    </xs:element>
    <xs:element name="TYPE" type="VODMLReference" minOccurs="0" maxOccurs="1">
     <xs:annotation>
      <xs:documentation>
       MUST reference the exact VO-DML type of the instance represented by the container
of the annotation element.
       Q: should we make minOccurs="1"?
      </xs:documentation>
     </xs:annotation>
    </xs:element>
    <xs:element name="OPTION" type="VODMLOptionMapping" minOccurs="0"
maxOccurs="unbounded">
     <xs:annotation>
      <xs:documentation>
       Identifies a mapping between an OPTION value and an EnumLiteral.
       Only to be used in PARAM/PARAMref/FIELDref.
       TODO make new subtype of VODMLAnnotation for these three contexts, extending ?
      </xs:documentation>
     </xs:annotation>
    </xs:element>
  </xs:sequence>
 </xs:complexType>
```

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. Regular expressions for mapping

Expressions for VOTable elements:

- ELEMENT   : any VOTable element that can be annotated, i.e. a GROUP, PARAM, FIELDref or PARAMref
- E ∊ F : VOTable element E is directly contained under F
- E ∉ F : E is not directly contained under F
- E ⊂ F : E has F in ancestry
- E ⊄ F : E has no F in ancestry
- ¬E/VODML/ROLE : E's VODML element has no ROLE
- G[R] : an element G for which R is true
- ...
- ⊃, ⊅
- ...

For VO-DML elements

- **`R`**∊ **`T`** : VO-DML Role **`R`** is defined on **`T`**
- **`R`**⊂**`T`** : VO-DML Role **`R`** is „available" on **`T`**, i.e. **`R`** is defined on **`T`** or a supertype of **`T`**
- **`R`**⊄**`T`** : VO-DML Role **`R`** is not available" on **`T`**, i.e. **`R`** is not defined on **`T`** or any supertype of **`T`**
- `. . .`

Mixed:

- E/VODML/TYPE ⇒ **`D`** : VOTable element E identifies VO-DML concept **`D`** as its VODML/TYPE
- E/VODML/TYPE = "a:b.c" : VOTable element E's VODML element has a TYPE with value "a:b.c"
- ...

Legal patterns

- {GROUP G| G ∊ VOTABLE & G/VODML/TYPE=”*vo-dml:Model*” & ¬G/VODML/ROLE} :
    - Model declaration
    - A GROUP G is contained directly under the VOTABLE element, it has a VODML element with TYPE set to “vo-dml:Model” and without ROLE.
- {GROUP G | G ⊄ TABLE & G ⊄ GROUP[VODML] & VODML/TYPE⇒**`ObjectType`** & ¬G/VODML/ROLE} :
    - standalone/singleton **`ObjectType`** instance
    - A GROUP G that is not contained in a TABLE, nor in a GROUP with a VODML annotation. It has a VODML element with TYPE indicating an **`ObjectType`** and no ROLE element
- {GROUP G | G ⊄ TABLE & VODML/TYPE⇒**`DataType`** & ¬G/VODML/ROLE} : standalone/singleton object type instance
-

Illegal patterns:

- {ELEMENT E| E ∊ GROUP[VODML] & ¬E/VODML/ROLE}
    - Any element inside a GROUP that has a VODML annotation, MUST have a VODML/ROLE
- {ELEMENT E| E[VODML] & E ⊄ GROUP[VODML] & ¬E/VODML/TYPE},
  {ELEMENT E| E[VODML] & E ⊄ GROUP[VODML] & E/VODML/ROLE}
    - Any element with VODML annotation, that does NOT have an ancestor GROUP with a VODML annotation, MUST have a TYPE and must NOT have a ROLE MUST have a VODML/ROLE
- {ELEMENT E| E[VODML] & E/VODML/ROLE & E/VODML/TYPE
  & E/VODML/ROLE ⊄ E/VODML/TYPE}
    - For an element with both a TYPE and ROLE annotation, the ROLE MUST be available on the declared TYPE.
-

Appendix D. 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.].