International
Virtual
Observatory
Alliance

Utype: A data model field name convention Version 0.4

IVOA Note November 6, 2009

This version:
http://www.ivoa.net/Documents/Notes/WD-Utypes-0.3-20090520.pdf

Latest version:
http://www.ivoa.net/Documents/latest/Utypes.html

Previous versions:
0.3

Editor(s):
Mireille Louys, ...???

Authors:
Mireille Louys, Jonathan McDowell, François Ochsenbein, Doug Tody, François Bonnarel, Alberto Micol, Gerard Lemson

Abstract

This document discusses the definition, usage and implementation of Utypes in the Virtual Observatory.

1 Status of this document

This document has been produced by the Data Model Working Group. It is still a draft.

Acknowledgements

Members of the IVOA Data Model Working Group, including representatives of the US NVO, Astrogrid, and the Euro-VO have contributed to the present draft. F. Bonnarel , Anita Richards and M. Louys acknowledge support from the European *EUROVO-AIDA* project.

Contents

1 Status of this document **1**

2 Introduction **4**
2.1 Scope of the document 4
2.2 Context and definition 4

3 What are Utypes for? **5**
3.1 Serialisation 5
3.2 Requirements for Utypes construction 7

4 Utypes Syntax **8**
4.1 Building-up the string 8

5 Data model re-use **9**
5.0.1 Canonical notation 11
5.0.2 Alternative Utypes representation 12
5.1 Short abbreviations for Utypes 12

6 Generating Utypes from UML data models via their XML representation **13**

7 How are Utypes documented? **14**

8 How are Utypes published? **15**

9 How are Utypes used? **15**
9.1 Publishing data to the VO 15
9.2 Naming metadata in VO protocols 16
9.3 Querying data bases 16

10 Conclusion **17**

A Appendix A: Utype serialisation example **18**

B Appendix B: VOTable serialisation example **19**

C Appendix C: Updates of the document **20**

2 Introduction

2.1 Scope of the document

This document is summarizing the practice adopted in the Virtual Observatory for naming and identifying data models elements. It defines the Utype concept, the syntax proposed to represent Utypes-lists in the VO , and finally illustrates how to use them.

2.2 Context and definition

In the field of astronomy, when two services or users need to share data, they can represent the necessary associated metadata using various data model compliant products. At the code level, they can share data model classes, then re-using the full Object Oriented modeling with classes and methods. In a more portable way, they also can exchange lists of metadata values labeled with names derived from the data model classes and relationships.

The advantage is to have homogeneous labels understandable by various data centers and to be able to first publish one's data in a VO understandable way, and then to compare metadata from data sets of different origins.

The Virtual Observatory provides protocols and interoperable applications in order to access, retrieve, analyze astronomical data. Services are principally based on the unified representation of metadata which are provided basically by data models for each domain: Observation, Spectra, Simulations, VOEvents, etc...

The metadata in astronomy are distributed using file formats like FITS, VOTABLE, which are rather flat representations for a data set or XML files which brings hierarchy. This is important for interoperable services and applications to be able to recognise and identify the role of one piece of metadata inside a VO Model. For example if we get (`SPATRES = 1.3 arcsec` in the FITS header of an observed spectrum or image, we would guess it is a spatial resolution and by browsing the Characterisation Data Model [3] consider that it can be expressed as a standard name or label or tag: *SpatialAxis.resolution.referenceValue* according to the structure of the SpatialAxis object in the model. This string is the name of the attribute used in the data model to represent this property of an observation.
Such a name, defined and understood in the context of a data model (here Characterisation) is called a Utype.

3 What are Utypes for?

The main goal of Utypes is to help to parametrize a data model, i.e. to describe all items in the model as a list of keyword-value pairs. This very simple flat representation of a model can be handled in various ways. We can take a list of metadata associated to a specific data set, parameterize it via Utypes, and store the resultant data in the fields of a table, in a parameter set, in a hash map in Java, or even in a FITS header (provided that unique FITS keyword names are associated with a corresponding Utype, as for example in the Simple Spectral Access protocol [1]).

Other semantic tags, like UCDs already exist to classify metadata, they can categorise physical quantities but are not precise enough to uniquely identify a piece in a data model. As long as new data collections appear with many different metadata organisation, the need to bind one piece of metadata (wavelength band-width in an optical observation) with its corresponding representation in an IVOA Data model (e.g. in SpectrumDM) is crucial to promote interoperability and make protocols and applications easier for the user.

Up to now, data models like SpectrumDM, CharacterisationDM, SimDB data model help to define, represent and manipulate metadata. They provide UML diagrams, XML serialisations and Utypes lists for the model classes.

Within the DAL WG, protocols such as SSA also makes use of Utypes. The SSA protocol version 1.04 has its own Utype serialisation attached in Appendix D: 'SSA Data Model Summary' of the standard document [1] .

3.1 Serialisation

Serialisation is a process that helps to represent collections of metadata in a transportable way -that is outside programs, and in compatibility with an IVOA Data Model. Models are built following object oriented programming principles. They are represented in UML (Unified Modeling Language), using mainly the class diagrams. From these classes descriptions, the developer can derive a library in Java or C++ or Python , that can operate on these classes, and re-use them for his/her own application. However in most cases, metadata circulate in the VO via files in specific formats: VOTable, FITS, XML or structured ASCII files. These are the places where Utypes can be used in order to map fields or elements in these files to data model items.

More precisely, there are 4 ways of using/exporting the data model structural organisation :

- implement the data model classes in an object oriented language. Then metadata associated with a 2D image for instance, are described by a

set of classes within the Observation DM. To publish the metadata values of such an image, one just need to instantiate the corresponding classes of the data model and use the setters and getters functions of these classes to load or export the attribute values.
- derive an XML schema from the data model. Every class and every class's attribute will be translated as an element. Nested classes will produce a tree like structure in XML. The metadata for let say a 2D image, is then an XML instance document following the XML schema structure.
- use a nesting strategy to bring back the hierarchy in a non hierarchical serialisation : VOTable, JSON [2], PARfile [] allow for that.
- re-use names of elements in the XML schema as keywords in a flat ASCII (keyword,value) list.

Considering an observation, – f.i. from the GOODS data set, to be published to the VO, how can we express its spatial, spectral, temporal, and photometric features? This is in the scope of the Characterisation Data Model (or in SSA Utype list as well). Various possibilities are available to describe such a metadata list for each data set:

1. use an XML instance document containing the whole tree of elements below the root element "Characterisation".
2. provide a (keyword, value) list with keywords mapping the leaves of the corresponding XML tree.
3. use a VOTable document, and attach to each main structure FIELD, PARAM, GROUP, or TABLE, a name from the data model elements. This representation is preferable for large collections of metadata chunks of similar structure.

See Appendix A, B, C for serialisations examples.

Every serialisation has its advantages:

- XML provides a hierarchy of nested objects and can directly use XML searching tools like XPath.
- The ASCII list of (keyword,value) pairs is the simplest most compact representation.
- VOTable encodes object nesting within GROUP elements and supports large collections of similar objects.

Hence all three should co-exist within the VO. The translation from one representation to the other should be bijective which implies that the Utype string must encode the nesting structure of the objects in the model. Whereas the graph structure of the UML class diagram is richer than the XML tree projection, the translation can still be organized both ways, provided some rules are adopted for the UML design as explained in Section 4.

3.2 Requirements for Utypes construction

The Utype purpose is essentially to point to the simplest atoms of a data model, i.e attributes within a class, so that it can be used in a pair like in (Utype,value). Composing a name for atomic elements is just using a string composition in most object programming language. For example, in Characterisation DM, pointing to the number of bins along the Spectral axis will be **SpectralAxis.numbins**. Most of attributes in data models are themselves classes, that can be browsed down in order to reach the lowest level of encapsulation and point to single value elements. This is the case for **SpectralAxis.coverage.Location.unit** with 2 levels of nesting.

If a data model gets more complex, like SimDB/DM or ObservationDM for instance, groups of classes involved in the same use-cases (functionalities) are identified and organised in packages. See the SimDB overview at : `http://www.ivoa.net/cgi-bin/twiki/bin/view/IVOA/IVOATheorySimDBDM` or the Observation DM sketchout at Fig 1. This should also be reflected in the string structure of Utypes.

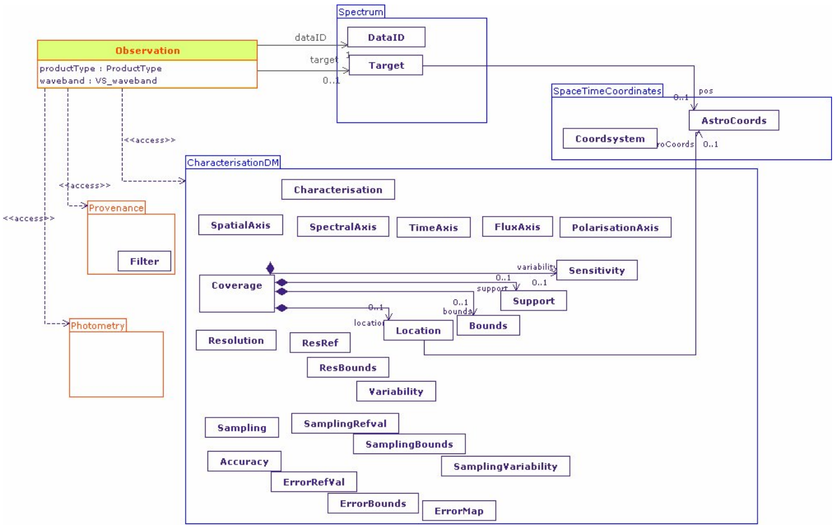

Figure 1: *Observation DM overview: This model include existing packages : Characterisation, Spectrum, Provenance, Photometry, and Spectrum and re-uses some classes of STC classes as base types for attributes.*

Because of the graphical structure of UML, linking classes with each other, a path inside the data model is not always unique. In order to be able to build-up Utypes names directly from a UML data model, we propose the following rules for UML design, and developed a Utype syntax from it.

4 Utypes Syntax

4.1 Building-up the string

questions to Gerard: corrections / suggestions?? The building up of Utypes has been discussed extensively within and between both Data Model WG and Theory IG and at various Interoperability meetings. Here is a proposed syntax we have agreed on for simple valued element.

The Theory interest group has tried to come up with a minimal, necessary set of rules to produce a string that uniquely represents any of the fundamental syntactic elements in the model. These rules are the following:

- Property names are unique in a Class. Note there are three types of properties:
  - An *Attribute* is a property the datatype of which is a value type (NOT an object type/class), though it need not be primitive but may be structured (i.e. have attributes of its own).
  - A *Collection* is a named, 1-to-many composition relation of a parent to a child class.
  - A *Reference* is a named, many-to-one shared association to another class.
- Class names are unique in a Package (name space).
- Package names are unique in either an enclosing parent package, or in the set of models adopted in the IVOA.

So a name like (in a pseudo regexp notation)

```
<model-name>:[<package-name>/]*<class-name>.<attribute-name>[.<attribute-name>]*
```

is a unique pointer to an attribute in a data model. Similarly

```
<model-name>:[<package-name>/]*<class-name>.<reference-name>
<model-name>:[<package-name>/]*<class-name>.<collection-name>
```

are unique pointers to the reference and collection properties of a class. *When classes are embedded, there may be attributes before the reference.How do we* *handle this?*

The rule allows for an arbitrary nesting of packages, which is necessary to ensure a unique encoding. Since attributes can be structured, we allow for chaining these until the final primitive attribute is reached, i.e one which carries a single value.

The reference name (resp. collection-name) is an explicit name for a link, or pointer to a target class, as shown in fig. 2 where *SimDB:simdb/experiment/Experiment.protocol*, is a reference to the protocol used to realize this particular Experiment.
to improve ...

In the Utype string construction, references and collections are NOT followed further. Only the pointing mechanism is expressed in the Utype. The referenced (target) class will be encoded normally and pointers will be implemented to it.

Each serialisation mode can support this :

- XML will use the ID/IDREF mechanism to set a link from the class to the referenced one provided the two connected classes are defined in the same document.
- VOTable applied the same mechanism.

Reference could be implemented in the (Utype,value) pair list, but XML and VOTable are much more convenient for that. Therefore the Utype list serialisation should be reserved for small sets of metadata consisting in single value attributes, as used in the various VO protocols.

Utypes for higher level, less primitive elements such as classes are obtained simply by not expanding attributes to the end. They are not useful for (keyword, value) lists serialisation, but are interesting in hierarchical VOTable serialisation, where by using a GROUP structure we can fully encode nested objects. See the VOTable Example in Appendix B.

All this works efficiently with a simple or complex self-explaining data model but in many case, a data model re-use classes from another datamodel as data types.

5 Data model re-use

However, in the VO there are common structures that are needed everywhere, like IVOA identifiers, or coordinates. Coordinates are defined in a separate

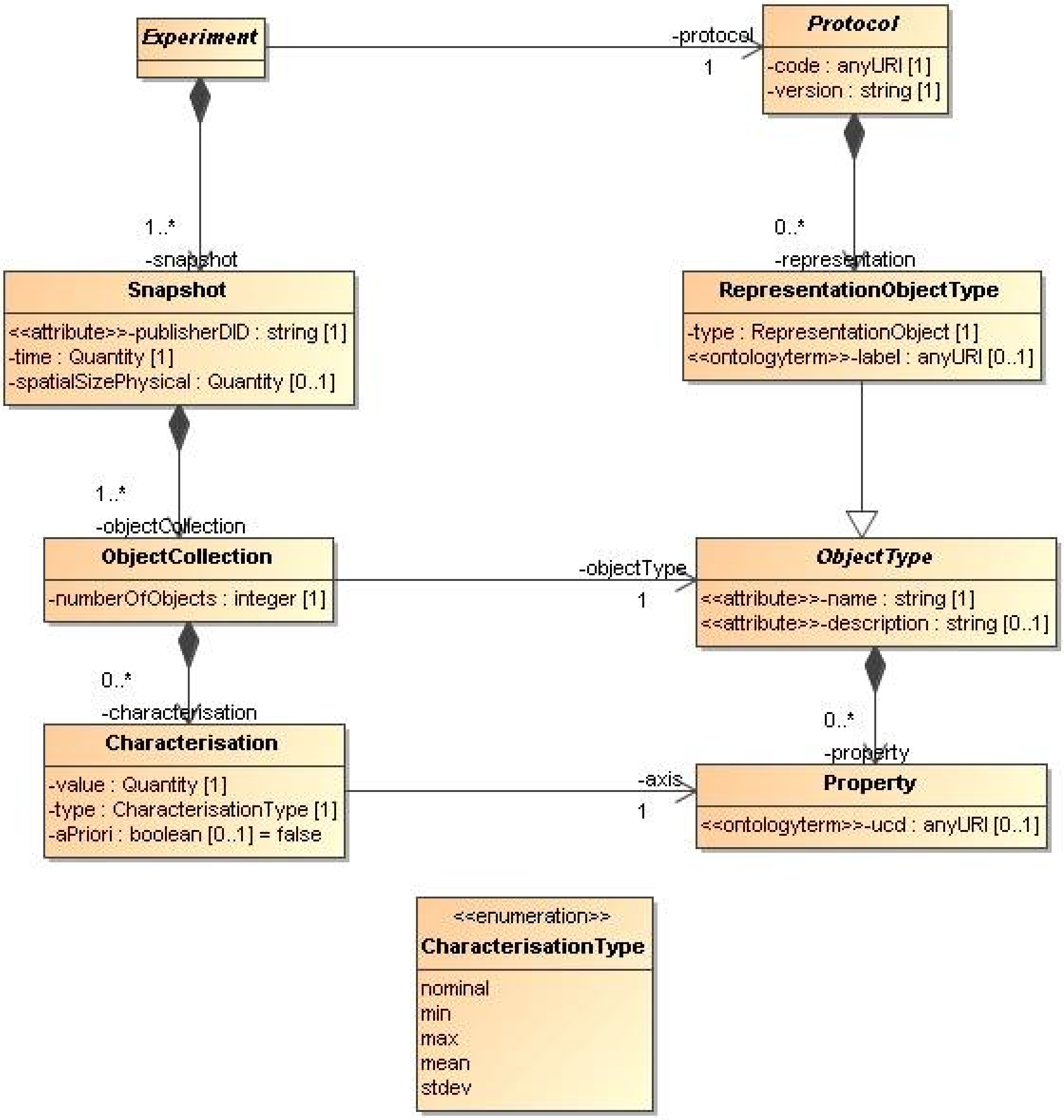

Figure 2: *UML diagram excerpt of the SimDB data model. This illustrates the reference mechanism between classes. Here ObjectType is the name of a class, that can be accessed from the ObjectCollection via a link or pointer called* `objectType`*. This means that the class ObjectCollection gathers objects whose types are described in the ObjectType class, allowing to consider any new types of objects.*

model: STC, http://www.ivoa.net/Documents/latest/STC.html and identifiers are standardised in at http://www.ivoa.net/Documents/latest/IDs.html.

In object oriented programs, classes of these packages are simply linked using libraries, and can then be used as types (primitive classes) for other models. For serialisation, we need an explicit mechanism to mention that attributes in a class re-use STC basic structures, for instance.

XML serialisation reusing other models are easy to build up as existing schemata can be linked together or imported. For instance the Characterisation data model imports STC elements which are then parsed using the XML name space mechanism. In the case of Utypes serialisation, there are two proposed strategies:

5.0.1 Canonical notation

This is the most explicit that allows various versions of the two associated models. Utypes are prefixed with their relative data model name space string and just concatenated using a specific delimiter as suggested in Fig. 3. Utypes would then be chained according to this pattern:

```
dm1:Utype1;dm2:Utype2
```

which means that entities named Utype2 in 'dm2' are re-used as atomic constructs inside Utype1 entities in 'dm1'. This notation helps to clearly identify the data model which each Utype belongs to.

```
<SpatialAxis>                              Corresponding UTYPE
<axisName>spatial</axisName>
<ucd>pos</ucd>
<unit>deg</unit>
<coordsystem id="TT-ICRS-TOPO"             <---->cha:SpatialAxis.Coordsystem
xlink:type="simple xlink:href="ivo://STClib/CoordSys#TT-ICRS-TOPO"/>
<coverage>
<location>                                 <---->cha:SpatialAxis.coverage.location
    <coord coord_system_id="TT-ICRS-TOPO">
        <stc:Position2D>
            <stc:Name1>RA</stc:Name1>
            <stc:Name2>Dec</stc:Name2>
            <stc:Value2>
            <stc:C1>132.4210</stc:C1>      <---->cha:SpatialAxis.coverage.location.Position2D.Value2D.C1
        <stc:C2>12.1232</stc:C2>
            </stc:Value2>
        </stc:Position2D>
    </coord>
</location>    …
</coverage>
</SpatialAxis>
```

Figure 3: *Correspondance between XML elements and Utypes: this example illustrates the similarities between the XML path reaching a leaf element and its Utype representation.*

dm1 and dm2 are name space prefix that point to the data model representation , for example the XML schema corresponding to the corresponding version of the model. The concatenating character ***; (semicolon)*** is not overlapping with any reserved characters of the VOTable standard, or XML, or uri syntax. In order to be able to use the Uri mechanism [**?**] described by Norman Gray, [`@,*,$,#,%`] not allowed in URI should be avoided.

The concatenation is supposed to happen only one time which means the right part after ';' is a kind of VO type described consistently and self sufficiently in one single data model. This makes the assumption that VO models are properly organised in nested packages and are cooperative enough to cover the whole field of astronomical metadata with a minimum of overlap.

5.0.2 Alternative Utypes representation

The canonical notation applied to the Characterisation data model provides very long Utype strings that are not appealing to the user and too long to be used as data base column names. If we consider for instance a specific version of the Characterisation data model whose classes integrates coordinates and regions from the STC v1.33 data model, we get a simpler notation by just browsing down the attributes chain as shown in the syntax section.

```
SpatialAxis.Coverage.location.coord;stc:Position2D.Value2D.C1
```

would simply become

```
SpatialAxis.Coverage.location.Position2D.Value2D.C1
```

Such a notation does not show the limit between the two models but is consistent with the XML schema import mecanism. Parsing the Utype string and resolving the name space will point to the specific version of the CharDM with the specific STC v1.33 data model version. This provides a fixed binding between the two data model versions. Although the string is not much shorten, it allows to pick up any single value in a data model instance and browse down the nested classes to build up the corresponding Utype string.

5.1 Short abbreviations for Utypes

From the building approach, Utypes are prone to be long due to the object oriented design that encourages nested classes and package re-use. However, even if it is a drawback for display in applications, long strings are easier to interpret by data providers and VO programmers, avoid ambiguities and foster uniqueness.

Inside an application, a data base or or a server, where Utypes are only machine-interpreted, alias to short names can be build and used internally. For instance a mapping table between Characterisation Utypes and local abbreviations are generated in the SaadaDB system [4].

6 Generating Utypes from UML data models via their XML representation

The syntax rules proposed in Section 4 above can be implemented from an XML schema representing the data model, using the XPATH mechanism [5] to build up a path from the root of the schema down to the finer grain elements corresponding to attributes' class in the model. XPATH is not directly used in Utype generation , but its properties are indirectly applied in the approach described here.

Suppose now that we have an XML schema fully mapping the UML model content, with all classes represented as elements in the model, nested elements for aggregation, references and basic types.

For the sake of clarity, we do avoid substitution groups and choice patterns and on the contrary prefer the XML extension mechanism. Such a rule helps to guarantee that for one XML element at any level, its name can be mapped to only one sub-structure and therefore allow for direct class encoding. Nested classes will be organized as XML trees, then browsing down the tree to leaves elements and concatenating the names provides a path which is similar to the Utypes construction mentioned in the previous section(cf 4).

In order to achieve a proper mapping from UML to XML serialisation, and derive object code or Utype list from the generated XML, some requirements on the style of UML design as well as the XML schema construction should be met.

- UML : For any association , each class connected should have a role name in order to clearly identify references. Template classes provide a same name for different typed structures and are difficult to translate in XML; hence they should be avoided.
- XML Classes, should be converted as XML elements and class attributes as included sub-elements. The XML attributes are more or less providing context for the XML translation and are not used to describe the data model structures(only valid for charac. simdb has a diff. strategy).

Most of the UML modeling commercial tools like RationalRose, MagicDraw, Objecteering , etc... have an internal XML representation of a UML model encoded in a proprietary XMI format. When simplifying this representation, one can apply XSLT transformation rules to directly generate output products like :

- an XML schema
- an example of XML document instance
- a Utype list with documentation
- a set of hyperlinked webpages for the datamodel documentation

Such an approach has been implemented with success by G. Lemson and L. Bourges in the Theory interest group. see http://volute...

UML allows various designs for a specific project and fully integrates the properties of graphs, with association links between classes while on the contrary XML emphasizes the hierarchy of elements. Therefore the translation is not straightforward. Some modeling rules should be imposed in UML design in order to simplify translation and produce robust XML schema and Utypes list. The Theory interest group [?] has tried to come up with a minimal, necessary set of rules to produce a string that uniquely represents any of the fundamental syntactic elements in the model. These rules are the following:

- Property names are unique in a Class. Note there are three types of properties: An Attribute is a property the data type of which is a value type (NOT an object type,/class), though it need not be primitive but may be structured (i.e. have attributes of its own). A Collection is a named, 1-to-many composition relation of a parent to a child class. A Reference is a named, many-to-one shared association to another class.
- Class names are unique in a Package (name space).
- Package names are unique in either an enclosing parent package, or in the Model (the root of all).

7 How are Utypes documented?

The documentation for a Utype is defined when the data model is build up and stored in the XMI representation of a UML Model. Most case tools provide a documentation generator that produces an HTML hyperlinked set of pages. These may contain just a set of few lines or a full illustrated text if necessary. N. Gray has proposed an URI generation function for each Utype in a DM, that could be used to point to the corresponding anchors of the on-line documentation of a data model.

8 How are Utypes published?

For each version of the VO data models, an explicit set of Utype strings is built up in an XML Schema enumerating the various Utypes strings. In VOTable documents or Utype-list, a name space definition should be included for Utypes validation.

Services /applications to describe, assign and parse all Utypes defined from a data model should be developed, similarly to the UCD tools available at http://cdsweb.u-strasbg.fr/UCD/ for instance. As a (training) example, the revised version of Characterisation DM, version 2.0 has a new XML schema and an updated set of Utypes available at http://ivoa.net/DM/UTypeListCharacterisationDM/UtypeListCharacterisationDM-V0.2-20090522.xsd... .

9 How are Utypes used?

9.1 Publishing data to the VO

Data Providers can use Utypes to label the metadata attached to their data collections. The process will be the following:

- select a data model which covers the domain of these data
- map proprietary metadata (FITS, Archive, Etc..) to VO DM Utypes
- generate metadata as serialised documents (VOtable, Utypelists, others?)

Different scenarios can be explored : to be developed: To publish data with the CharacterisationDM-v1.11 , one can use the CAMEA VO Tool (http://eurovotech.org/twiki/bin/view/VOTech/CharacEditorTool) to check the Utype assignation, and verify if the Utype serialisation is compliant to this model. other strategy?

At the data collection level , tools have been developed to help for keyword mapping from FITS keywords to Utypes list: Here is a list of the first tools developed for that:

- FITS to DAL interface or data model Utypes:
- MEX (ESO) DAL interface link...
- DM-Mapper (ESA) DAL interface link...
- Interactive mapping tool (CDS) (prototype) link... This tools takes a data model description and helps the data provider to interactively build a map table from FITS keywords to Utypes.

Such a tool is under development and should be stabilized and tested for different data models. It would help data providers to map their metadata to a standardized VO Utype description.

9.2 Naming metadata in VO protocols

The SSA query response consists of a number of fields, identified by Utypes, grouped into component data models of the form ¡component-name¿.¡field-name¿. This is used in the Simple spectra access (SSA) protocol with a specific list of 'hand-carved ' keywords list representing objects structure . See Appendix D of the Simple Spectral Access Protocol V1.04 standard document at http://www.ivoa.net/Documents/latest/SSA.html
.

Similarly the SLAP protocol defines its own set of Utypes in the Appendix D of the Simple Spectral Line Access Protocol V0.9 standard document(http://www.ivoa.net/Internal/IVOA/SpectralLinesListDocs/WD-SLAP-0.9-20090518.pdf
).

The protocols generally use Utypes pointing to leaves of a data model:

9.3 Querying data bases

Queries in ADQL or SQL use column names to ask for information. For a data base to be compliant with a data model, only the mapping between existing columns and Utypes must be defined. Unfortunately Utypes strings may be longer than the allowed length for a column string content in the Data base systems, therefore Utypes cannot be used directly in queries. Using a mapping table allow to build up a service where: -

1. the client application ask a server for its list of supported metadata and Utypes
2. the server exposes the metadata
3. The user selects the metadata he/she requires by browsing the Utypes and the documentation.
4. the client translates each Utype in the query into a column name and submits the query
5. the server parses and resolves the query and sends back the results columns
6. the client translates each column name in Utypes when possible and display the results.

Such a scenario is interesting as if offers a general vocabulary to the user , whatever the data base content and needs few steps of re-engineering.

10 Conclusion

Utypes are useful to convey the role, the structure and the normalized name for each piece of metadata involved in a service or a protocol. It is an important factor in interoperability. A compromise between long descriptive strings and usability has been found in developing simple mapping mechanism at the client side.

References

[1] Tody D. Et al. Simple spectral access protocol. *http://www.ivoa.net/Documents/latest/SSA*, 2007.

[2] D. Crockford. The application/json media type for javascript object notation (json). *http://tools.ietf.org/html/rfc4627*, 2007.

[3] Louys M. et al. Utype list for the characterisation data model v1.11. *http://www.ivoa.net/Documents/Notes/UTypeListCharacterisationDM/UtypeListCharacterisationDM-20070522.pdf*, 2007.

[4] Michel L. et al. Saada : an astronomical data base generator. *http://amwdb.u-strasbg.fr/saada/spip.php?article32*, 2009.

[5] W3C. Xml path language (xpath) 2.0. *http://www.w3.org/TR/xpath*, 2007.

A Appendix A: Utype serialisation example

$i$nclude a simbd or snap simulation serialisation??

B Appendix B: VOTable serialisation example

```
<VOTABLE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xmlns:http://www.ivoa.net/xml/VOTable/VOTable-1.1.xsd"
xmlns:ssa="http://www.ivoa.net/xml/DalSsap/v1.0" version="1.1">

<RESOURCE type="Results">
<DESCRIPTION>
Sample of a getMetadata query response on a Simple Spectrum Access (SSA) service
</DESCRIPTION>
...
<!-- characterization metadata: Char.FluxAxis -->
<PARAM ID="FluxAxisUcd" name="OUTPUT:FluxAxisUcd" datatype="char"
utype="ssa:Char.FluxAxis.Ucd" arraysize="*" value="" >
<DESCRIPTION>UCD for flux</DESCRIPTION></PARAM>
<!-- characterization metadata: SpectralAxis -->
<PARAM ID="SpectralAxisUcd" name="OUTPUT:SpectralAxisUcd" datatype="char"
utype="ssa:Char.SpectralAxis.Ucd" arraysize="*" value="" >
<DESCRIPTION>UCD for spectral coord</DESCRIPTION></PARAM>
<!-- characterization metadata: Char.*.Coverage -->
<PARAM ID="TimeLocation" name="OUTPUT:TimeLocation" datatype="double"
ucd="time.epoch" utype="ssa:Char.TimeAxis.Coverage.Location.Value"
unit="d" value="" >
<DESCRIPTION>Midpoint of exposure on MJD scale</DESCRIPTION></PARAM>

<PARAM ID="TimeExtent" name="OUTPUT:TimeExtent" datatype="double" ucd="time.expo"
utype="ssa:Char.TimeAxis.Coverage.Bounds.Extent" unit="s" value="" >
<DESCRIPTION>Total exposure time</DESCRIPTION></PARAM>

<PARAM ID="TimeStart" name="OUTPUT:TimeStart" datatype="double"
ucd="time.expo.start"
utype="ssa:Char.TimeAxis.Coverage.Bounds.Start" unit="d" value="" >
<DESCRIPTION>Start time</DESCRIPTION></PARAM>

<PARAM ID="TimeStop" name="OUTPUT:TimeStop" datatype="double" ucd="time.expo.end"
utype="ssa:Char.TimeAxis.Coverage.Bounds.Stop" unit="d" value="" >
<DESCRIPTION>Stop time</DESCRIPTION></PARAM>

<PARAM ID="SpatialLocation" name="OUTPUT:SpatialLocation" datatype="double"
ucd="pos.eq" utype="ssa:Char.SpatialAxis.Coverage.Location.Value" arraysize="2"
unit="deg" value="" >
<DESCRIPTION>Spatial Position</DESCRIPTION></PARAM>

</RESOURCE>
</VOTABLE>
```

Figure 4: *Identifying pieces of a data model: SSA service. Here is a short extract of the Query response of an SSA protocol implementation. A VOTable document is returned, each of metadata being mapped to a Utype name in the SSA Utype data model.*

C Appendix C: Updates of the document

- version 0.3 to 0.4
  - introduce canonical and alternative notations
  - update fig.1 and fig.2