



International
Virtual
Observatory
Alliance

UTYPE: A Data Model field name convention

Version 0.2

IVOA Note 2007-09-03

Draft Version 0.2 Rev 1

<http://www.ivoa.net/Documents/WD/Utype/Utype-0000>

Latest version:

<http://www.ivoa.net/Documents/latest/Utype.html>

Previous versions:

None

Note: ivoa.net links are not yet active; the above are placeholders.

Editors:

Jonathan McDowell

Contributors:

Jonathan McDowell, other people TBD

Abstract

This document defines the syntax, semantics and usage of the UTYPE and UFI concepts. This draft replaces one dated 2007 May 10, and incorporates feedback received since then.

Status of this document

This is an IVOA Note. It may evolve into a WD.

This document has been developed with support from the National Science Foundation's <http://www.nsf.gov/> Information Technology Research Program under Cooperative Agreement AST0122449 with The Johns Hopkins University, from the UK Particle Physics and Astronomy Research Council (PPARC) <http://www.pparc.ac.uk>, and from the European Commission's Sixth Framework Program <http://fp6.cordis.lu/fp6/home.cfm> via the Optical Infrared Coordination Network (OPTICON), <http://www.astro-opticon.org>.

The **Virtual Observatory (VO)** is general term for a collection of federated resources that can be used to conduct astronomical research, education, and outreach.

The **International Virtual Observatory Alliance (IVOA)** (<http://www.ivoa.net>) is a global collaboration of separately funded projects to develop standards and infrastructure that enable VO applications.

Contents

1	Introduction and Motivation	4
1.1	UTYPE definition and syntax	4
1.2	Namespacing and scope	5
1.3	Combining data models	6
1.4	Instances and UFI's	6
1.5	Value fields	6
1.6	Serializations: XML	7
1.7	Serializations: VOTABLE	7
1.8	FITS	8

1 Introduction and Motivation

The DM group has a requirement to identify the parts of a data model in various contexts. A data model consists of low level information (fields with values) and a structure organizing them (classes). If we ignore the structure, we get a long checklist of all the information data providers might need to describe their data. Each piece of individual information, and each structural grouping (class) is called a data model field, and is given a name, which we call a UTYPE. Example:

`Spectrum.Target.Name`

The UTYPE (uniform type) attribute was first introduced in the context of VOTABLE to map the flat VOTABLE structure to a structured data model. The VOTABLE is an XML document with a fixed header-plus-table schema; to infer a data model structure we must impose extra clues on it. Each class or attribute of the model is assigned a string, allowing software to recreate the structure of the underlying schema. We use the phrases 'data model field' and 'utype' interchangeably. In the VOTABLE context, we have for example

```
<PARAMETER name="foo" utype="Spectrum.Target.Name" value="3C 273"/>
```

UTYPEs are also used in the FITS serialization of the Spectrum model to map FITS columns to the model via a TUTYPn keyword.

UTYPEs differ from UCDs. A UCD describes which physical concept is being used; a UTYPE describes its role in a given data model. Sometimes these are very closely linked and sometimes not. For example, in the Spectrum model, the y axis of the spectrum always has the utype `Spectrum.Flux.Value` but may have any of a large number of possible UCDs describing flux, surface brightness, luminosity or other photometric quantities.

UTYPEs may be thought of as a 'path' (like a directory path) into the data model. However, UTYPEs do NOT identify a particular instance of a data model element. There may be multiple instances of a given data model field in an instance document. To specify one of those instances uniquely, we need a language which can qualify the utype by querying on the field values, somewhat analogous to the XPATH syntax in XML. The IVOA Unique Field Identifier (UFI) syntax will provide this functionality.

1.1 UTYPE definition and syntax

- The utype is a string whose syntax is a series of tokens separated by dots (periods), optionally prefixed by a colon-terminated namespace:

`[namespace:]token.token.token...`

- The namespace defines the data model being used and the token items are a hierarchical series of data model objects or elements, with the highest level class to the left.
- A utype token is a syntactic element of a utype. Tokens may be nested to arbitrary depth, connected by periods. The existence of a utype token1.token2.token3 implies the existence of the higher level utype token1.token2 as a class containing the element token3.

For example, the utype

`Spectrum.Target.Name`

indicates an element Name which is a member of an object called Target, which itself is a member of a data model called Spectrum. In other words, the tokens imply a 'has-a' hierarchy.

- We can distinguish two types of utype: those which specify something with a simple data type (numeric, string) and those which specify complex objects aggregating more than one simple data type.
- When the context (position in hierarchy) is clear, we may refer to the utype by the rightmost token; this is called a 'simple utype' (e.g. 'Name'). Simple utypes are not currently recommended for use in VO XML documents; instead the 'fully qualified utype' (e.g. 'Spectrum.Target.Name') should be used.
- Listing all the utypes for a data model that specify a simple data type will define the information content of a data model.
- A utype token may contain only letters, digits and the underscore character. It may not contain punctuation characters such as ., /, *, dash, etc.
- utypes are case-insensitive. However, we recommend so-called CamelCase style.

1.2 Namespacing and scope

If a document contains the utype 'Target.Name', what is Target? Which data model defines Target? What version of that data model?

We can maintain a single namespace of utypes, as is done for UCDs, so that Spectrum.Target.Name is uniquely predefined. The set of top-domain utypes like Spectrum is then the set of data models in use by the IVOA. But this does not allow any way to specify the different versions of a given data model.

In XML the natural thing to do is to define a namespace

```
xmlns:foo="http://ivoa.net/xml/SpectrumModel/spec1.01.xsd"
```

and use a namespace prefix

```
<name utype="foo:Target.Name">
```

This allows us to mix different versions of different models: one might have spec:Target.Name and survey:Target.Name in the same document with different meanings.

QUESTION: No real differences between Spectrum.Target.Name and foo:Target.Name; the latter is more in the xml spirit, the former is simpler to parse (A. Micol - simple searching on utypes).

QUESTION: Top level scope.

The top level data models currently using utypes include Spectrum and Char. In addition, the Spectrum model is considered to be an aggregation of other models (Target, DataID, etc.) which may be used as top level items. A 'fully qualified' utype is one which begins with one of these top level models: thus Spectrum.Target.Name is fully qualified, but Name alone is not.

1.3 Combining data models

There are also cases where one data model uses another one. For example, in the Characterization DM, `Char.CharacterizationAxis.Coverage.Location` is the utype of an object whose data type is STC Coordinate. This complex object may contain a unit field with utype `Coordinate.CValue.Unit`. How do we express the fact that this particular STC Coordinate instance is being used inside the Characterization DM with specific semantics (it's the coordinate of the Coverage Location)?

Within STC itself, the `CoordFrame` model has in some cases instances of `Coordinate`, and one can end up with a utype `Coordinate.Resolution.PosAngle.Value` in the context of a `RedshiftFrame.CustomRefPos.Coordinate`. The simplest thing here may be to just extend the higher level model automatically, generating the immense utype

```
RedshiftFrame.CustomRefPos.Coordinate.Resolution.PosAngle.Value
```

Another approach would be to adopt the 'noun-adjective' method of UCD, and have:

```
Coordinate.Resolution.PosAngle.Value;RedshiftFrame.CustomRefPos
```

This form emphasizes that we have an instance of a field from the `Coordinate` data model, and its context is a field from the `CoordFrame (RedshiftFrame)` model.

DISCUSSION REQUIRED HERE.

1.4 Instances and UFIs

There may be several instances of the same utype within a document. For example, there may be more than one `Char.CharacterizationAxis` distinguished only by the UCD of the element. The CDS group (and in particular F. Bonnarel) have proposed an extended utype syntax for queries and links, in which the syntax

```
token1.token2[attribute2=valueA].token3.token4[attribute4=valueB]
```

is interpreted to mean that we are looking for that instance of an element with utype `token1.token2.token3.token4` that has `attribute4=valueB` and is inside an element with utype `token1.token2` that has `attribute2=valueA`. Others have suggested that the full XPATH syntax should be adopted. Since the syntax for UFIs is still under discussion, this document will not specify it; we mention it to emphasize and clarify the distinction between UFIs and utypes.

An alternative proposal, from Roy Williams, is to require that the serializations (from his context, presumably in VOTABLE) impose the requirement that utypes are unique within a table, and a repeated utype is handled by splitting the data into more than one tables. In a complicated data model, an instance document would then have very many tables, and the problem is now into one of specifying the relationships between the tables - but basically unchanged at the abstract level.

1.5 Value fields

A very common case in our data models is a 'decorated value', a class where one element is the main value we are interested in and the others are peripheral metadata. For example, we might have a utype `Foo.Error` with member utypes `Foo.Error.Value`, `Foo.Error.Unit`, `Foo.Error.UCD`. A convention has been suggested in which applications which are looking for simple data types would accept `Foo.Error` as equivalent to `Foo.Error.Value`. In this convention, the utype token `Value` would be treated specially, in that when mapping a utype against a model, a supplied utype `U` would match both `U` and `U.Value`.

1.6 Serializations: XML

The abstract data model field names are the defining elements of the data model. We make the model concrete via serializations: rules for writing the model in various formats.

Each DM must have a reference XML Schema, with an element or attribute corresponding to each data model field. **The reference schema should normally use the utype tokens (simple utypes) as the schema element names**, so that `Spectrum.Target.Name` would be written in the instance document as

```
<Spectrum><Target><Name>foo</Name></Target></Spectrum>
```

However, sometimes practical considerations may make the schema slightly different for the abstract model.

In an XML document the information content may be represented in several ways: either as the **contents** of elements or attributes:

```
<answer>42</answer>
<distance unit="furlongs">100</distance>
```

or as the **names** of the elements themselves:

```
<SpatialAxis>
<SphericalCoordinate>
<CartesianCoordinate>
```

as opposed to

```
<Axis type="Spatial">
<Coordinate type="Spherical">
<Coordinate type="Cartesian">
```

In some cases it may be preferable to make an explicit 'type' string-valued element in the abstract model, but represent it by the typing and subclassing mechanism in XML. So, a field `Coordinate.Type` in the model with values 'spherical', 'cartesian' might map to the existence of two different types `<SphericalCoordinate>` and `<CartesianCoordinate>` in the XML. Having it as a string in the abstract model allows us to use the latter as a 'checklist' of information that the data provider needs to generate.

Another example in which the abstract model and the XSD may differ is when the XSD schema is used for purposes other than describing the data structure and information content. For instance, the STC schema is used to force automatic validity checking of instances, in ways that add complexity which is relevant only in the XML representation.

1.7 Serializations: VOTABLE

In a VOTABLE, the XML structure of a document is given by the VOTABLE schema and not the model schema. The UTYPE attribute is used to capture the model structure in the document:

```
<PARAM name="foo" utype="Spectrum.Target.Name" value="3C 273"/>
<FIELD name="flux" utype="Spectrum.Data.Value"/>
```

For user readability and for uniqueness of namespace the GROUP element can be used to reflect the model:

```

<GROUP utype="spec:Data.SpectralAxis">
  <FIELDref ref="Coord"/>
  <GROUP utype="spec:Data.SpectralAxis.Accuracy">
    <FIELDref ref="BinLow"/>
    <FIELDref ref="BinHigh"/>
  </GROUP>
  <PARAM name="Resolution" utype="spec:Data.SpectralAxis.Resolution"
    ucd="spect.resolution;em.wave" unit="Angstrom" datatype="float" value="14.2"/>
</GROUP>

```

In the above case, the fully qualified utype is somewhat redundant and one might consider a scheme in which elements inherit the utype of the parent group, so that Resolution here would just have utype="Resolution", inheriting the prefix "spec:Data.SpectralAxis" from its parent GROUP. However this requires more parsing by the client and we are not currently supporting that scheme.

1.8 FITS

In the Spectrum data model we introduced a FITS representation of the model. In this representation, a fixed set of keywords was used to map to the utypes of the model, with arbitrary names chosen to fit within the 8-character limit of FITS keyword names.

However, for the binary table columns considerations of compatibility and user readability drove us to more flexibility in the column names (TTYPEn values). We therefore introduced a new keyword TUTYPn to hold the utype of column n. If TUTYPn is present, its value is the utype of column n; for example in the Spectrum model we might have

```

TTYPE4='WAVE_ERR'
TUTYP4='Data.SpectralCoord.Accuracy.StatError'

```