VO-URP and UTYPEs

I discuss here my view how an approach ala VO-URP can help in our UTYPEs discussion.

"Executive" Summary of Main Points

The VO-URP approach to UTYPEs proposes a concrete definition what it means for a `utype`[1] attribute to be a "pointer into a data model". It does so by stating that the value of a `utype` should be chosen from the set of identifier assigned to certain types of elements in a data model document that is well formed according to a standard meta-model. This meta-model defines the data modeling *concepts* we can use in constructing data models and that one can expect to show up in non-standard serializations. The meta-model thereby also defines the concrete concepts to use in "meta-DM" discussions, discussions where we talk of data models in general, not about specific examples such as STC, or Characterization. The UTYPEs discussion is such a meta-DM discussion, and the VO-URP approach can help it by providing the potential target for "pointers into a data model".

By stating that UTYPEs are "only" identifiers, we can reduce discussion on their format; basically limiting that to a question how the data model context can be attached in a VOTable or other serialization format.

What is very important though is that this approach also offers a formal definition of the different types of model constructs a serialization of a data model can contain. It allows us to make the discussion as to how how these constructs can be represented in, or mapped to a VOTable more formal and concrete. It can provide the framework to discuss how UTYPEs can help describe this mapping by defining the types of things in a data model a UTYPE can point at, and how this assignment is to be interpreted and possibly restricted.

As a side effect, a common meta-model also offers a formal way to compare models, maybe to combine and reuse them. Some data models have done so, but the proper use has not been formalized. This has come up in our discussions, but needs not be part of a UTYPE spec I htink.

In this document I suggest that the most important point in the discussions of the UTYPEs Tiger Team is to define how UTYPEs are to be used and what the interpretation is of a UTYPE assigning a particular data model concept to a particular VOTable element.

I propose that we can quickly deal with UTYPE formatting issues, and start discussing a restriction on the way utypes are used to indicate data model membership of annotated VOTable elements. In particular I propose as one possibility to restrict this completely to GROUP elements and their contained components. We can discuss constraints and requirements on the contents of such annotations. I also propose that the UTYPEs that can be assigned directly to TABLE and FIELD elements be used for special circumstances of a complete ORM-like mapping only. Allowing them to be mixed with annotations through GROUPs would greatly increase the required complexity of client interpreters and increase the possibility of redundancy, errors and inconsistencies.

[1] When talking about the UTYPE concept, I use upper case, when referring to a concrete `utype` attribute I use `courier`.

I finally make a proposal how the UTYPEs should be formatted when used in VOTable, namelythat they consist of a prefix that identifies the model via a model declaration INFO element at the root VOTable element, a semi colon and then an identifier that identifies the element in the corresponding data model document. The **vo-urp** meta-model contains a specific utype element for this usage.

Statement of the problem

I believe the main problem to have complicated the discussions in the UTYPE tiger team is that we have no clear definition of what problem we are trying to solve. In fact one may argue we were asked to come up with a definition of the problem, because we should define use cases and prioritize them.

I think there is consensus that the following question(s) MUST be answered:

MQ: What can we do with the `utype` attributes in VOTables?
I.e. what values can be assigned to `utype` attributes, what is their interpretation and are there any rules how the values can be assigned?

This question can be generalized, and other more or less related questions can be asked, but I strongly believe that if we can come up with an acceptable (proposal for a) specification that addresses this question we have succeeded. I think a specification limited to the usage of UTYPEs in VOTable likely provides a suggestion for similar ones dealing for example with FITS files or other data formats. And as a side effect of our deliberations we may propose to the IVOA exec the need for some further specifications as well[2].

For the answer to the above question to be acceptable and for us to be able to address it in a meaningful way, we need to (be allowed to) make some assumptions, and allow some constraints. I believe some of these are:

1. UTYPEs are intended to give meaning to elements in the VOTable they are assigned to.
2. UTYPEs SHOULD do something different from the `ucd` attribute in those cases where both can be assigned to the same VOTable element.
3. UTYPEs MUST "point into a data model". But we need to define what precisely this means.
4. The solution SHOULD (be able to) use of the results of the IVOA data modeling working group.
5. The solution MAY include the usage of other features that the VOTable format provides, such as GROUP-ing, FIELDref-s etc.
6. UTYPEs identify the roles the different VOTable elements play in the mapping from VOTable to Data Model, i.e. they identify the instances in a meta-model mapping.

I think these are reasonable and in particular can lead to a useful end result. I do *not* believe that we MUST or even SHOULD obey the following constraints for our solution to be acceptable

1. UTYPEs are serialisations of the data model.
2. UTYPEs MUST have a structure that allows us to infer the data model from the format alone.
3. UTYPEs must be XPaths

[2] E.g. "how do we specify data models?, how can one data model use another one?, how do we map data models to other formats?".

4. Data models must/can be treated as trees
5. …

These suggestions have come up in our discussion and may need to be discussed further, but I will not do so here in great detail. I think they arise from questions that all come down to

Q: Is the structure of UTYPEs important?

A lot of discussion has centered on the format of the UTYPE strings. This is likely induced by the fact that existing UTYPE lists seem to reflect the structure of the data models for which they were defined. Early discussions of UTYPE were also often confusing "pointer into data model" with "query path into a (XML) data model instance document identifying a particular element". The latter were coined by Jonathan McDowell as UFIs, and are *not* discussed here. The theory group's SimDM effort provided a grammar for constructing UTYPEs in an automated manner from the data model elements. This was ever only meant though as an automatable algorithm to provide unique UTYPE values for the elements, not as a suggestion that this MUST be followed by all UTYPE libraries, and most definitely not to serve as a replacement for the actual data model itself.

I think main objections to spending too much time on the structure of UTYPE-s, are the following

1. There is a solution to much/all of the supposed usages of UTYPEs that can be largely independent on their structure (apart from prefix possible).
2. Doing without data model that are defined using some accepted standard approaches requires instead a new design of how UTYPEs can define and represent data models.
3. A typical VOTable will not contain all possible UTYPEs from a given model, so how can one infer the full model if not through some externally defined document? If so, why not do so using a standard methodology?

But we may need some more true use cases where structure is important.

In the rest of this document I will focus on proposing an approach for answering the main question **MQ** above, whilst accepting the relevance of the list of assumptions following it.

VO-URP: meta-model and faithful mapping

VO-URP[3] is a framework that assists in handling complex data models.
In particular it assists in defining data models by offering a modeling language for expressing the data model in a simple XML format. We refer to this language as **vo-urp**[4]. This modeling language can also be seen as a meta-model, instances of it are data models[5]. Elements form the meta-model are presented in Appendix B of the Simulation Data Model (SimDM), and reproduced partially in an Appendix to the current document [TBD].

Important for the discussion is that to almost every element in a **vo-urp** model one MUST associate a <utype> element that must be unique. By using the infamous grammar (also first introduced in

[3] Currently, VO-URP is maintained in the **vo-urp** project in Google Code at http://code.google.com/p/vo-urp/.
[4]The current version of the XML schema defining the language (and the one used in Simulation Data Model) can be found at http://ivoa.net/Documents/SimDM/20120503/uml/intermediateModel.xsd
[5] For a more formal approach from the *Object Management Group* to different levels of models see the *Meta Object Facility* at http://www.omg.org/mof/.

SimDM) to derive a value based on the type of element and certain uniqueness constraints in the meta-model itself, VO-URP ensures uniqueness of the <utype>. Hence the value of the <utype> can be used as an identifier of the corresponding element[6].

VO-URP adds to this meta-model an automated mapping procedure that translates a *logical* data model expressed in **vo-urp** to various alternative machine and human readable *physical* representations[7] such as XML Schema, Relational Database Schemas, HTML documentation, Java class definitions etc. The way VO-URP achieves this is by assuming a particular mapping of the data model components as specified by the **vo-urp** meta-model, to the meta-model components of the target representation. I.e. VO-URP implements various meta-model mappings.

The mappings from logical to physical representations created by VO-URP are intended to be basically[8] lossless. I.e. the information in the XML or relational schema is 1-1 w.r.t. the information in the underlying **vo-urp** model. Consequently complete or *faithful* serializations of the model can be made through (sets of) XML documents, or in a relational database, or through in-memory Java objects. These serialization formats MIGHT be added to an IVOA data model specification. Clients and servers MIGHT communicate with each other using these physical representations. For example a DAL service MIGHT return an XML document representing the PhotDM in its specified form, rather than a VOTable. Clients can interpret the XML and do something with it. Or a TAP service can be specified that has the specific relational representation as TAP_SCHEMA. In these cases, where a physical representation is part of the DM specification, one does strictly speaking not need additional annotation, such as UTYPEs, to identify the relation between elements in the physical representation and those in the logical representation.

However one still MIGHT use annotation and it may well be useful for an otherwise faithful TAP_SCHEMA where for some reason the names of columns or tables are not identical to the names in the model. Indeed the TAP_SCHEMA that VO-URP derives from a **vo-urp** data model *does* contain values for the `utype` attributes. The role of the UTYPE there is to identify that a certain TAP_SCHEMA::table represents a certain vo-urp::ObjectType, a certain TAP_SCHEMA::column a certain identified vo-urp::attribute or a TAP_SCHEMA::key corresponds to a certain vo-urp::reference This is achieved simply by giving the `utype` attribute the value of the appropriate <utype> element in **vo-urp**.

In the next section I will argue that the same approach can also work for non-standard, non-faithful mappings, and that precisely here most of our effort should be targeted. Though in principle the meta-model that is used can be designed from scratch, VO-URP can assist in the current effort by providing a meta-model that already includes utypes as an identifying mechanism, has been given quite some thought and has been shown to work quite well for our SimDM data modeling effort. It even allowed us to generate full web applications from the model alone. Of course I don't claim **vo-urp** is perfect or that it can be applied without further change. But at least it gives a concrete

---

[6] In VO-URP we have a separate identifier attribute for each element as well. This should be an xsd:ID, so that it can be used in xsd:IDREF attributes also part of the meta-model. In the original approach in VO-URP, where the **vo-urp** is derived from an XMI representation of a UML model, the XMI:id attribute is used to give this id its value.

[7] For some explanation on logical and physical data model Google can point you for example to http://www.agiledata.org/essays/dataModeling101.html and many other sites.

[8] Maybe not completely, as some choices are made during the generation such as mapping of data types etc, and some elements simply cannot be mapped completely, such as references to XML schema.

example of such a meta-model which we can use in investigating what it takes to support partial mapping, how UTYPEs can be used and what constraints we may need to define.

VO-URP and UTYPEs: Custom mapping

For concreteness we will from now on assume the target representation to be a VOTable. What really is required is that the target representation must have some structure, a meta-model of its own really. It must consist of well defined, identifiable elements to which a `utype` attribute (or element?) can be associated. This `utype` we will for concreteness assume to identify an element from **vo-urp**, though another similar meta-model could serve as well[9]. There is currently no faithful mapping from **vo-urp** to VOTable (though it may be interesting to define one). But in any case we need to consider incomplete, "unfaithful" representations.

In VOTable a `utype` attribute[10] can be assigned to elements of various types. This assignment, by assumption identifies some **vo-urp** element in a data model. The interpretation of the assignment depends both on the type of the VOTable source element and the **vo-urp** target element. The following matrix shows the possible matchings. It lists vertical the VOTable elements that have a `utype` attribute in their definition, horizontal the **vo-urp** elements that have a <utype> element in theirs. An "x" indicates what I think are possibly meaningful associations, and which I will discuss below. But this is open for argument.

Table 1 Matrix of possible UTYPE assignments for VOTable elements.

vo-urp ⇨ / VOTable ⇩	Model	Package	Class/ ObjectType	DataType	Attribute	Collection	Reference	Container	ID
INFO	?								
FIELD					x		x	x	x
PARAM					x		x	x	x
GROUP			x	x	x	x	x	x	x
FIELDref					x		x	x	x
PARAMref					x		X	x	x
TABLE			x	x		x	x		
RESOURCE		x	x						

We discuss these possible assignments next. [TO BE COMPLETED]

- `INFO`
  Unclear what an assignment of a UTYPE to an Info element could mean. It is basically a name/value pair. An predefined info element with UTYPE "vo-urp::Model", with name="<prefix>" value="<datamodel –url>" could be used to link prefixes to URLs of the data model document to which they refer. But see below on a use for Group for that purpose. Could we insist that Info elements should *not* be used for mapping of general data model constructs, as we have other elements for that?
- `FIELD`
  - **Attribute**: indicates that a column or a parameter carries the value of a particular attribute.

[9] One might use UML/XMI (whichi is very complex) or RDB, Java,or XML Schema (which have as disadvantage that they are very implementation oriented).

[10] Note that the choice of an XML schema attribute restricts the possible form a UTYPE could assume. It must be a string following XML attribute constraints. Cannot for example have explicit inner structure as an element could have had! Any structure must be based on string parsing.

Should one also expect that a group has been defined corresponding to the object type containing the attribute? Or that the table containing the field has a UTYPE pointing to appropriate ObjectType?

- **Reference**: indicates that a table column or a parameter has as value an identifier referencing an object of the data\type of the reference. One would expect that elsewhere in the VOTable a `Group` or `Table` has been assigned a UTYPE corresponding to the referenced type.
  Should one also expect that a group has been defined corresponding to the object type defining the reference?
- **Ccontainer**: indicates that a table column or a parameter carries an identifier of an object containing the object. Similar considerations apply as for a UTYPE identifying a reference.

- `PARAM`
  - **Attribute**:
    - When in GROUP
    - When in a TABLE …
    - When in a RESOURCE …
- `GROUP`
  If a Group element is assigned a UTYPE, we could insist it MUST map to an ObjectType, or a (structured)DataType. One would expect that contained FIELDRef, PARAMref and Param elements would have UTYPE pointing to an Attribute, Reference, Container or ID element of the corresponding Type. As soon as a FIELDref is included one would expect that the Group indicates that the table contains instance of the corresponding type in its rows.
  We might also insist that GROUP elements are used to indicate the various Model-s that are used in the VOTable and which prefix is used for these.
  - **Type: ObjectType, (structured) DataType**:
    One would expect all contents of the Group to be compatible with the contents of the type. FieldRef-s would indicate attributes or references or a container. A contained Group would correspond to an object in a collection of the type.
  - **Model**
    A Group can also be used to indicate model information, including the UTYPE prefix to be used. The Group itself would have UTYPE “vo-urp:model” or so, where the “vo-urp:” prefix plays the role of the “xmlns:” prefix in namespace declarations in XML documents. This group can contain any important model information, in particular its URI and the prefix to be used. (see below for some examples). Note, I think the prefix should be left free to the context, and not be predefined in the model.
  - :
- `FIELDref`
  Main question is what the difference is/can be between the UTYPE on a FIELDref and the one on the FIELD it is referring to.
  Should this have a UTYPE? What is the **.vo-urp** construct this could correspond with? Could it be different form the UTYPE of the `Field` that is actually being referenced here?
  Alternatively we could potentially ONLY use GROUPs and FIELDrefs to give mapping information on TABLEs and FIELDs. And not put that info in these elements themselves.
- `PARAMref`
  Similar comment as for FIELDref

- `Table`
  In object relational mapping a table generally is identified with a class. Same can be done here:
  - **Type: ObjectType**
    One would expect contents of the Table to be compatible with the contents of the type. Field-s would indicate attributes or references or a container.
  -
- `Resource`
  If we see a RESOURCE as a grouping of TABLEs one might link it to a pcakge. On the other hand it might also be an instance of a single class, PARAMs could then be attributes or references and contained TABLEs could correspond to collections of the contained class.
  -

TBC

References

1. Norman Gray, *UType questions* 2009 http://nxg.me.uk/note/2009/utype-questions/
2. Norman Gray *UType Proposals* 2009 http://nxg.me.uk/note/2009/utype-proposals/
3. Markus Demleitner *etal Referencing STC in VOTable Version 2.0* 2010 http://www.ivoa.net/Documents/Notes/VOTableSTC/20100618/NOTE-VOTableSTC-2.0-20100618.pdf
4. Ochsenbein, F. *etal* 2009 *VOTable Format Definition Version 1.2* http://www.ivoa.net/Documents/VOTable/20091130/REC-VOTable-1.2.html
5.

Appendix: vo-urp meta-model and UML Profile

Important for the discussion of UTYPE-s is the data modeling language that VO-URP uses to express models in a machine readable XML format. This language can be viewed as a meta-model, in the same sense as the relational model (consisting of schemas, tables, columns, foreign keys etc) is a meta-model for relational databases, or XML Schema is a meta-model for XML documents.
We refer to this language as **vo-urp** and it is expressed as an XML schema in its own right and is also supported by a UML Profile that we used in the SimDM efforts. **vo-urp** contains the relevant information of the model in a form that is *much* simpler than for example the official XMI format. A**.vo-urp** document can be derived from an XMI document, under the condition the UML is extended with a suitable UML Profile. An example of a script that generates the **vo-urp** representation from the XMI version exists in VO-URPO as well and was used to generate the **vo-urp** representation[11] of the MagicDraw UML version[12] of the Simulation Data Model[13].

Adopting a common language for our DM activities is useful in that it gives us standard concepts to use when discussing data models in a general, abstract way, rather than discussing a particular data

[11] http://ivoa.net/Documents/SimDM/20120503/uml/SimDM_INTERMEDIATE.xml
[12] http://ivoa.net/Documents/SimDM/20120503/uml/SimDM_DM.xml
[13] For a PNG version of that model see http://ivoa.net/Documents/SimDM/20120503/uml/SimDM_DM.png

model. For example this is the case when discussing mapping of data model to an alternative representations. Just like in general discussions on Object Relational mapping strategies one may talk about mapping Classes to Tables, Attributes to Columns, Relations to Foreign Keys, a common language would allow us to state that a VOTable FIELD can map to Attribute, GROUP to DataType etc[14]. I.e. the view of VO-URP is that we should aim to discuss mapping of *meta-models*. So we do not *have* to have separate discussions how to map STC to VOTable, or SED, or PhotDM etc.

To be able to do so more concretely we first must introduce an example meta-model for which we use **vo-urp**. The main modeling elements we use in the meta-model were listed and explained in Appendix B of the SimDM document. They are all explicitly defined in the **vo-urp** schema. For the present discussion we focus on the following elements, the meaning of whichi I summarise, for more details see the SimDM doc. In square brackets the concept representing the element in UML:

- Model [Model]: one or more packages with types.
- Package [Package]: A collection of type definitions and possibly sub-packages. Provides name spacing and grouping of connected type definitions.
- ObjectType [Class]: A complex type, consisting of attribute definitions and relations to other ObjectTYpes. Have an implicit *identity*, in contrast to ValueTypes.
- *ValueType* [abstract, has subtypes]: concept that does not have its own identity, but can be relatively complex.
    - DataType [DataType]: can have attributes, e.g. Pos3D=[x,y,z]. May have reference(s) to ObjectTypes[15].
    - PrimitiveType [PrimitiveType]
    - Enumeration [Enumeration]
- Collection (and its inverse, Container) [binary association end of composite relation]: Indicates a composition relation between a parent and contained child ObjectType .
- Reference [binary association end of shared relation, possible cardinality 0..1 or 1]: Indicates a shared “uses” relationship between a Object- or DataType and a target ObjectType.
- Attribute [Property not defined through relation, its data type must be a ValueType]: property of Object- or DataType with datatype a ValueType, i.e. *not* an ObjectTYpe (references are for that).

vo-urp defines these all as XSD complexType definitions. These types all extend an abstract *Element* type from which they inherit a *utype* attribute. The utype should be unique in a model. One way to achieve that is to derive their value from the model itself according to the grammar we defined in the SimDM document, and taken over in Mireille’s first draft of the UTYPEs document. But this is not required for the present discussion!

The meta-model embodied in **vo-urp** of course is more than just this listing of data modeling elements. It also prescribes in full detail what further properties they have and how they may/must be used in combination. Hence an Attribute has a *name* and a *datatype* and must be part of an

[14] It may still be useful in certain specific cases to make explicit features available for special models. E.g. consider the case of the old (?) COOSYS element in VOTable and its possible mapping to a specific STC concept. But there we do not need UTYPEs, and that is what we are discussing here.

[15] Not yet in SimDM version of VO-URP. But very useful for example to define an STC SkyPosition consisting of [longitude, latitude, reference to coordsys] or so.

ObjectType or a Datatype; an ObjectType can *extend* a base ObjectType, and can have a *Collection* relation with another ObjectType etc.

Element

All elements mentioned below are specialisations of UML Element.

```
<xsd:complexType name="Element" abstract="true">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0"/>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
        <xsd:element name="utype" type="xsd:string" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation>
              UTYPE of the element.
              Utility element, could be obtained from data structure, but then the ObjectType UTYPE
              generation rule needs multiple locations of implementation.
              Now (in principle) only in utype.xsl.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="xmiid" type="xsd:ID">
    <xsd:annotation>
      <xsd:documentation>
      This is the xmi:id of the corresponding element in the source XMI representation.
      It is used in xmiidref attributes in TypeRef and attributes to provide an explicit
      lookup functionality when we need it and to link back to the original XMI document.
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
        <xsd:attribute name="otherutype" type="xsd:string" use="optional">
          <xsd:annotation>
            <xsd:documentation>
              Reference to an external UTYPE.
              Can be assigned to each model element.
              Is meant to indicate that the concept represented by the element is
               . equivalent with
               . imported from
               . similar to
               . ...
              the concept represented by the referenced element in the external model.
            </xsd:documentation>
          </xsd:annotation>
        </xsd:attribute>
</xsd:complexType>
```

Model

This is the root of the complete model, contains all packages, classes etc. Also contains any imported profile.

vo-urp

```
<xsd:element name="model">
  <!-- patch so JAXB understands that this element is an XmlRootElement -->
<xsd:complexType>
  <xsd:complexContent>
    <xsd:extension base="Element">
      <xsd:sequence>
        <xsd:element name="lastModifiedDate" type="xsd:long"/>
        <xsd:element name="author" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="title" type="xsd:string" minOccurs="0" maxOccurs="1"/>
 ...
       <xsd:element name="package" type="Package" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
</xsd:element>
```

Package

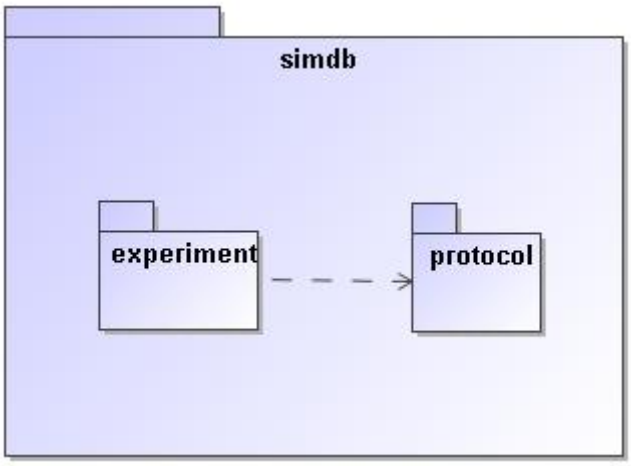

Figure 1: This figure shows a package "simdb" that contains two other packages. Of these the experiment package depends on the protocol packages, which is indicated by the dashed arrow. See Error! Reference source not found. **for the somewhat more complex package structure used in SimDM.**

A package groups related elements such as class definitions and possibly sub packages. Packages can depend on each other (indicated by the dashed line), which means that elements in one package can use elements in the target package in their definition. This relation is transitive. A package is similar to an XML namespace and in fact we map UML packages to XML namespaces in the XML schema mapping for the model described in **Error! Reference source not found.**.

vo-urp

```
<xsd:complexType name="Package">
  <xsd:complexContent>
    <xsd:extension base="Element">
      <xsd:sequence>
        <xsd:element name="depends" type="PackageReference" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="objectType" type="ObjectType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="dataType" type="DataType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="enumeration" type="Enumeration" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="primitiveType" type="PrimitiveType" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="package" type="Package" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>

</xsd:complexType>
```

Type

Base class of different type definitions.

vo-urp

```
<xsd:complexType name="Type" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="Element">
      <xsd:sequence>
        <xsd:element name="extends" type="TypeRef" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="abstract" type="xsd:boolean" default="false" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Class[16]

Figure 2: A Class is a rectangular box, with the name of the class in boldface.

Classes are the fundamental building blocks of a data model. A Class represents a full-fledged concept and is built up from properties and relations to other Classes. An important feature of Classes as opposed to DataTypes (see below) is that instances of Classes, i.e. objects, have their own, explicit identity[17]. That is, we want to assign an explicit identifier to each particular usage of this concept, for instance here to distinguish between various Experiment instances.

vo-urp

```
<xsd:complexType name="ObjectType">
  <xsd:complexContent>
    <xsd:extension base="Type">
      <xsd:sequence>
...
        <xsd:element name="attribute" type="Attribute" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="collection" type="Collection" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="reference" type="Reference" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
...
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

ValueType

A ValueType represents a simple concept that is used to describe/define more complex concepts such as Classes. ValueType-s are, in contrast to Classes not separately identified. They are identified by their value. For example an integer is a value type; all instances of the integer value 3 represent the same integer.

In this profile ValueType-s are only represented using specialised examples.

Attributes (see below) must have a ValueType as their datatype.

---

[16] Also referred to as *ObjectType*

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

vo-urp

```
<xsd:complexType name="ValueType" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="Type">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

PrimitiveType

PrimitiveTypes are the simplest examples of ValueTypes. They are represented by a single value only. A set of PrimitiveTypes is predefined in the IVOA profile (see Figure 3).

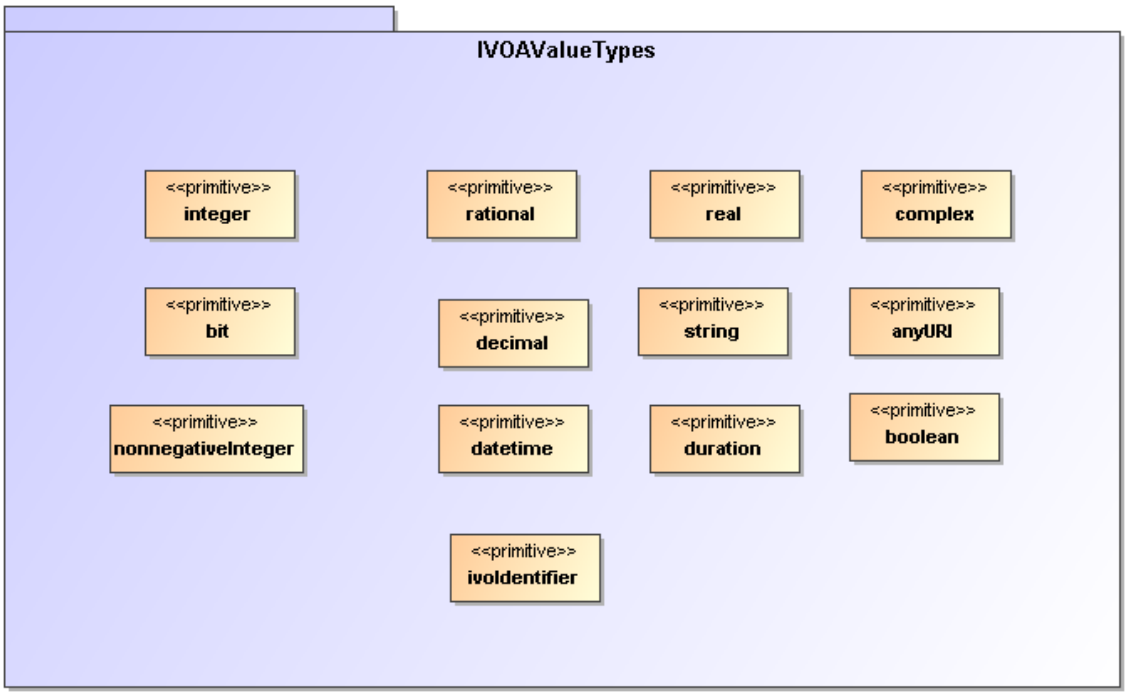

Figure 3: The PrimitiveTypes that are predefined in the IVOA profile.

vo-urp

```
<xsd:complexType name="PrimitiveType">
  <xsd:complexContent>
    <xsd:extension base="ValueType">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

DataType

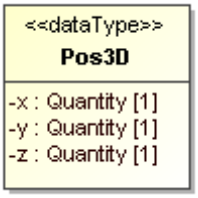

Figure 4: Example of a structured datatype:Pos3D represents a position in 3D space and is defined using x, y and z attributes. The DataType symbol is distinguished from the Class by the <<dataType>> stereotype.

A DataType is a ValueType that has more structure than a single value. This structure is modelled using Attributes, just as on ObjectTypes.

vo-urp

```
<xsd:complexType name="DataType">
  <xsd:complexContent>
    <xsd:extension base="ValueType">
      <xsd:sequence>
        <xsd:element name="attribute" type="Attribute" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Enumeration

Figure 5: An enumeration is indicated by a box with the name of the enumeration and the list of valid literals.

An Enumeration is a ValueType that is defined by a list of valid values. These are the only values that instances of this data type can assume.

vo-urp

```
<xsd:complexType name="Enumeration">
 <xsd:complexContent>
   <xsd:extension base="ValueType">
     <xsd:sequence>
       <xsd:element name="literal" maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="value" type="xsd:string"/>
             <xsd:element name="description" type="xsd:string" minOccurs="0"/>
```

```
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Attribute

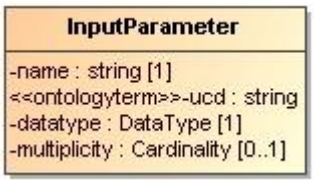

Figure 6: An attribute is indicated by a line with a name, a datatype, an indication of the multiplicity and possibly a stereotype.

An Attribute is a Property of a type (object type as well as structured data type). An attribute's data type is always a Value type, not an object type. For object type properties one should use a Reference.

vo-urp

```
<xsd:complexType name="Attribute">
  <xsd:complexContent>
    <xsd:extension base="Element">
      <xsd:sequence>
        <xsd:element name="datatype" type="TypeRef"/>
        <xsd:element name="multiplicity" type="Multiplicity"/>
        <!-- should next be on TypeReference? the constraints restrict the type after all. -->
        <xsd:element name="constraints" type="Constraints" minOccurs="0"/>
        <xsd:element name="skosconcept" type="SKOSConcept" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Inheritance

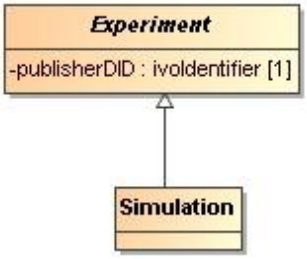

Figure 7: Inheritance is indicated by a line with an open arrow from a subclass to its base class.

Indicates the typical “is a” relation between the sub-class and its base-class (the one pointed at). In this profile we do not support multiple inheritances.

Relation

Base concept of colelctions and references. Indicate relations between two ObjectTypes, or between a dataype and an objecttype:

vo-urp

```
<xsd:complexType name="Relation" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="Element">
      <xsd:sequence>
        <xsd:element name="datatype" type="TypeRef"/>
        <xsd:element name="multiplicity" type="Multiplicity"/>
        <xsd:element name="subsets" type="xsd:string" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="xmiidref" type="xsd:IDREF"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Collection

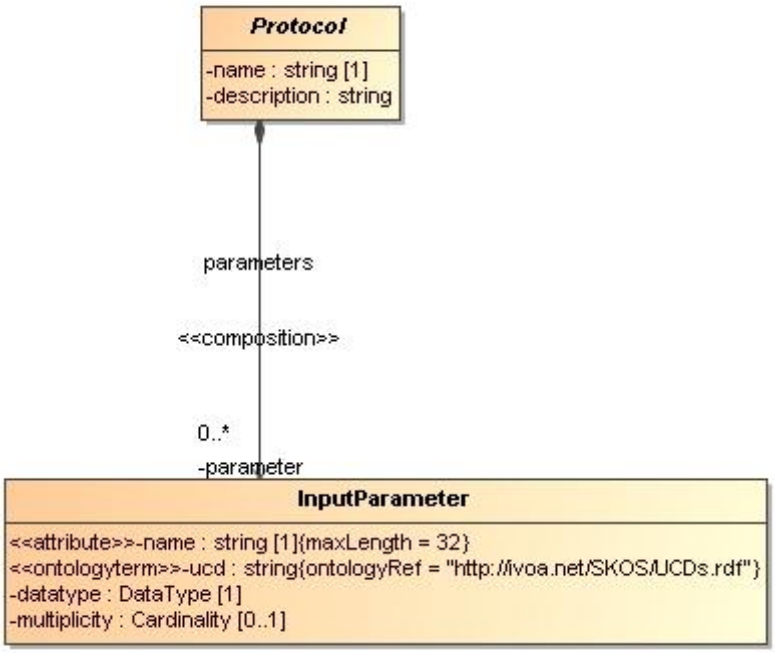

Figure 8: The line with the filled diamond on one end and an arrow on the other indicates a (parent-child) composition relation, or collection, between the parent, on the side of the diamond; and the child, on the other side.

This relation indicates a composition relation between one parent, container object and 0 or more child objects. The life cycles of the child objects are governed by that of the parent.

In UML a composition relation is represented by a binary association end.

vo-urp

```
<xsd:complexType name="Collection">
  <xsd:complexContent>
    <xsd:extension base="Relation"/>
  </xsd:complexContent>
</xsd:complexType>
```

Reference

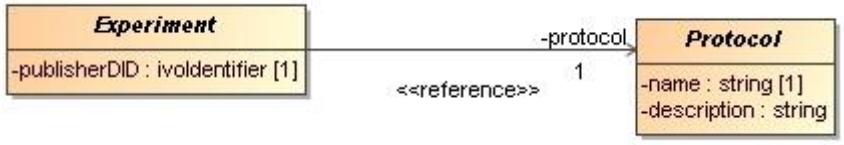

Figure 9: A Reference is represented by a line connecting a class with another, referenced, class with an arrow on the referenced class. Note, the <<reference>> stereotype indication is not required.

This is a relation that indicates a kind of usage, or dependency of one object on another. It is in general shared, i.e. many objects may reference a single other object. Accordingly the referenced object is independent of the "referee". In our profile the cardinality cannot be > 1.

For implementing the Reference in UML we use a shared, navigable binary association end.

vo-urp

```
<xsd:complexType name="Reference">
  <xsd:complexContent>
    <xsd:extension base="Relation"/>
  </xsd:complexContent>
</xsd:complexType>
```