Transformations in the VO Data Model

David Berry (dsb@ast.man.ac.uk), 8th September, 2003.

This document presents some ideas for the description of transformations within the context of the VO data model. It is presented in response to a request for discussion on this subject made by the Data Modelling working group at the IVOA interoperability meeting at Cambridge.

Transformations may be needed to describe the relationships between different world coordinate systems and/or data value systems associated with a resource. For instance, the data values in a sub-mm data set may be described in different systems such as flux, effective antenna temperature, *etc.* Pixel positions in a CCD image can be described in terms of pixel coordinates, focal plane, coordinates, sky coordinates, *etc.* It would be advantageous for each data set to include descriptions both of the dimensionally distinct systems in which the values and positions in the data set can be represented, and also of the transformations between these different systems.

The scheme described below allows for the transformation of both scalar and vector values. It is based on the experience gained over the past 8 years or so with the Starlink AST library (see http://axp0.ast.man.ac.uk/~dsb/ast/ast.html).

This document does not include any formal description of how to represent a transformation in XML, RDF, *etc*. These descriptions can follow if necessary.

The Mapping Class:

For the purposes of this document, a *transformation* is a machine-readable recipe for converting a vector representing a single position within some (unspecified) input coordinate system into a corresponding vector in another unspecified output coordinate system. The base class is referred to as a *Mapping* and represents a pair of transformations – one being the *forward* transformation (from input to output coordinate system) and the other being the *inverse* transformation (from output to input coordinate system). A Mapping must define at least one of these transformations - optionally one transformation may be undefined.

Note, an important distinction is made in this document between a Mapping, and the coordinate systems which describe the inputs and outputs of the Mapping. A Mapping is simply a recipe which describes a sequence of mathematical operations to be applied to a supplied N-dimensional vector, in order to create a corresponding output vector. A Mapping does not include any description of the input and output coordinate systems and does not make any assumptions about their nature. For instance, the recipe represented by a particular Mapping may be "multiply all elements of the input vector by 3.5 and then subtract 6 from the second element". Such a recipe says nothing about the nature of the input or output vectors.

However, a Mapping is only of any use when it is used to transform positions from some known coordinate system into another. So there must be some way of specifying the properties and nature of these coordinate systems, such as "effective antenna temperature in units of Kelvin", "galactic longitude and latitude in units of degrees". This document does not address the issue of how such coordinate system descriptions should be stored, or the issue of how to form the connection between a Mapping and the objects which describe its input and output coordinate systems. Arnold

Rots “Space Time Coordinates” schema is of relevance here, as are the “Frame” and “FrameSet” classes in the Starlink AST library.

An important consequence of formally dissociating a Mapping from its input and output coordinate systems is that it is then possible to treat Mappings as “black boxes” which can be combined in any fashion without any reference to what the inputs and outputs represent. Thus complex recipes can be formed by combining simple atomic steps.

The Mapping class encapsulates the properties which are common to all Mappings, but does not itself define any transformations. For this reason, it is an abstract class which cannot be instantiated. A wide range of sub-classes of Mapping can be defined, each of which implements specific forms of transformation. It is these sub-classes which are instantiated.

The properties of a Mapping are as follows:

Name	***Type***	***Description***
Nin	integer	The length of the input vector (*i.e.* the number of axes in the input coordinate system).
Nout	integer	The length of the output vector (*i.e.* the number of axes in the output coordinate system).
TranForward	boolean	Does the Mapping define a forward transformation?
TranInverse	boolean	Does the Mapping define an inverse transformation?
Inverted	boolean	Has the Mapping been inverted?

Notes:

1. The *Nin* and *Nout* properties need not be equal. For instance, a Mapping which converts from spherical (longitude,latitude) to Cartesian (x,y,z) coordinates on a unit sphere will have 2 inputs but 3 outputs.
2. The *Inverted* property is used to modify the behaviour of the Mapping by inverting its normal “forward” direction. For instance, an “inverted” ExpMap implements the same forward transformation as a LogMap, and vice-versa.

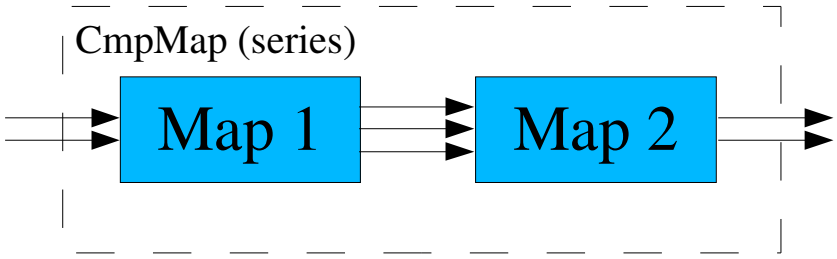
The principal methods of a Mapping are:

Name	***Purpose***
Transform	Use a Mapping to transform a set of input positions into corresponding output positions
Differentiate	Use a Mapping to find the rate of change of a specified output with respect to a specified input at a given input position
Simplify	Create a new Mapping which is a simpler form of a given Mapping
Invert	Invert a Mapping by toggling its *Inverted* property

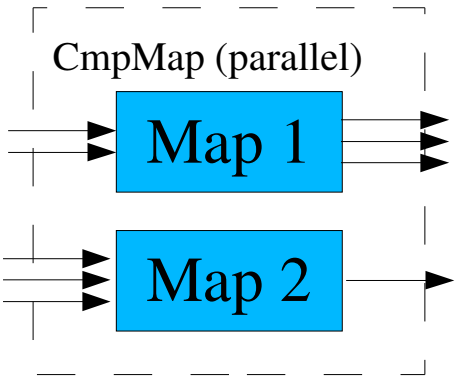
plus property accessor methods of course.

Compound Mappings:

An important sub-class of Mapping is a CmpMap. A CmpMap is a compound Mapping which allows two component Mappings (of any class) to be connected together to form a more complex Mapping. This connection may either be "in series" (where the first Mapping is used to transform the coordinates of each position and the second mapping is then applied to the result), or "in parallel" (where one Mapping transforms some subset of the input coordinates for each position and the second Mapping simultaneously transforms the remaining input coordinates).



Above is an example of a series CmpMap which combines a (2-input,3-output) Mapping with a (3-input,2 output) Mapping to form a (2-input,2-output) CmpMap.



Above is an example of a parallel CmpMap. The values for input axes 1 and 2 are transformed by the first component Mapping to produce output axes 1, 2 and 3. The values for input axes 3, 4 and 5 are transformed by the second component Mapping to produce output axis 4.

The properties of a CmpMap are as follows:

Name	***Type***	***Description***
Map1	Mapping	The first component Mapping
Map2	Mapping	The second component Mapping
Series	boolean	Are the component Mappings combined in series? (otherwise they are in parallel)

Since a CmpMap is itself a Mapping, it can be used as a component in forming further CmpMaps. Mappings of arbitrary complexity may be built from simple individual Mappings in this way. It is easy to create overly-complex CmpMaps – that is, CmpMaps in which the effect of some of the Mappings cancel out. A simple example would be a CmpMap which combined a (potentially complex) Mapping in series with its own inverse – the total effect of the CmpMap would be

equivalent to a unit Mapping . Whilst legal, this is bad practice for several reasons: the CmpMap will take longer to evaluate, it will introduce greater rounding errors, and will require a larger description. For this reason, the *Simplify* method of the Mapping class is important: it creates a new Mapping from a supplied Mapping by removing any redundant steps in the supplied Mapping.

Atomic Mappings:

Appendix A lists a large collection of concrete sub-classes of the Mapping class, each of which implements a specified simple mathematical operation or function (*e.g.* "x+y", "x-y", "x*y", "x/y", "x**y" "sin(x)", "max(x,y)", *etc.*). Complex Mappings can be created by combining these atomic Mapping within CmpMaps. Some of these atomic Mappings require extra properties to hold parameter values used in the mathematical operation.

Packaged Mappings:

It is possible to represent almost any complex transformation by combining the atomic Mappings listed in Appendix A into nested CmpMaps. However, the evaluation of such transformations (*e.g.* using the *Transform* method) could be made more efficient by having dedicated Mapping classes to represent commonly used specialised transformations. An example is matrix multiplication. A possible approach to providing a Mapping to do matrix multiplication would be to combine multiple AddMaps, MultMaps and PermMaps into a CmpMap which would do the required matrix multiplication. Another approach would be to define a whole new sub-class of Mapping, a *MatrixMap,* which encapsulates the values of the matrix elements directly. The *Transform* method of the MatrixMap class would do the matrix multiplication directly, using the matrix elements stored in the MatrixMap. By comparison, the *Transform* method of the equivalent CmpMap would invoke the *Transform* methods of its component Mappings recursively until the underlying atomic Mappings were reached. This interpretive process is bound to be slower than the direct approach implemented by the Matrixmap class.

So it would be advantageous to define a collection of "packaged" Mappings for commonly required complex operations, but at the same time leaving open the possibility of building up complex Mappings from atomic Mappings in order to cover more esoteric cases which are not common enough to justify their own dedicated Mapping class. Other examples of useful "packaged" Mappings would be various forms of spherical projections, conversion to and from different celestial, spectral or temporal coordinate systems, pin-cushion distortion, spherical to Cartesian conversion, *etc*. Of course, these packaged Mappings could themselves be combined together within a CmpMap to form even more complex Mappings.

Appendix A: A suggested list of atomic Mappings

The following is a list of concrete sub-classes of the Mapping class which implement various mathematical operations. *Nin* and *Nout* are the number of input and output values for the Mapping.

A "*" indicates that any positive number may be used. A *Nout* value of "nin" means that *Nout* must be equal to *Nin* (in this case, each output will usually be a function of the corresponding input).

The list is not intended to be exhaustive, it is just a list of examples. Many obvious Mappings have been omitted for brevity.

Name	***Nin***	***Nout***	***Function***	***Extra Properties***
PermMap	*	*	Re-arrange input values into a different order, optionally adding or removing inputs or outputs.	Description of the axis rearrangement
LutMap	*	*	Calculate output values from a look-up table.	The look-up table values
AddMap	*	1	Adds all input values together	
SubMap	2	1	Input 2 minus input 1	
MultMap	*	1	Multiply all input values together	
DivMap	2	1	Input 2 divided by input 1	
PowMap	2	1	Input 2 is raised to the power of input 1	
ShiftMap	*	nin	Add a given value onto each input value	The shift for each input
ZoomMap	*	nin	Multiply each input by a specified value	The scale for each input
RecipMap	*	nin	Take the reciprocal of each input	
SqrMap	*	nin	Square each input	
SqrtMap	*	nin	Square root of each input	
RaiseMap	*	nin	Raise each input to a given power	The power for each input
CosMap	*	nin	Cosine of each input	
SinMap	*	nin	Sine of each input	
TanMap	*	nin	Tangent of each input	
LogMap	*	nin	Natural logarithm of each input	
ExpMap	*	nin	Exponential of each input	
MaxMap	*	1	Maximum of all inputs	
MinMap	*	1	Minimum of all inputs	
TestMap	3	1	Output is equal to input 2 if input 1 is non-zero. Otherwise output is equal to input 2.	
GtMap	2	1	Output is 1 if (input 1 > input 2) and zero otherwise.	
LtMap	2	1	Output is 1 if (input 1 < input 2) and zero otherwise.	

Name	*Nin*	*Nout*	*Function*	*Extra Properties*
EqMap	*	1	Output is 1 if all inputs are equal and zero otherwise.	tolerance for equality
OrMap	*		Output is 1 if any of the inputs are non-zero.	