DAP Data Model Specification
DRAFT

James Gallagher*, Nathan Potter†, Tom Sgouros

Printed: November 3, 2003
Revision: 1.64

Contents

*The University of Rhode Island, jgallagher@gso.uri.edu
†Oregon State University, ndp@coas.oregonstate.edu

1 Introduction

This document describes the *Data Model* of the OPeNDAP [1] Data Access Protocol (DAP) Version 4.0. The data model is the framework—the set of data types and representations—with which the DAP represents the contents of a data source. The data model also encompasses the objects in which these data types are encoded: the replies with which an OPeNDAP server responds to requests for data.

This document contains several sections.

Section 2 (page 4) defines what is meant by a data source and describes the kinds of variables a data source may contain. These variables include the basic *atomic* types, and the more complex *constructor* types.

Section 3 (page 11) All DAP variables can have *Attributes* to describe them further. Representation of this information is described here.

Section 4 (page 12) Variables in a data source may be sampled by means of a constraint expression, defined in this section.

Section 5 (page 17) describes the interaction between a client requesting data and the server providing it.

Section 6 (page 18) The variables in a data source are given their persistent representation in the data objects defined here. This is the representation used to communicate between a server and client.

The DAP is independent of the lower-level communication protocols used to implement it, such as HTTP, FTP, GridFTP, Telnet, *et cetera*. DAP implementations currently exist for several communication protocols. HTTP is the most commonly-used implementation, and a separate document, ***DAP Web Services Specification***, is available from the OPeNDAP project to specify its use. The OPeNDAP project does not issue specifications for other protocols. This is left to the groups making the implementation.

A note about terminology. In this document, the words ”client” and ”server” are meant only to imply a program making a request via the DAP and another program making a reply, respectively. The two programs are probably running on machines remote from one another, but this is not essential. Like WWW clients and servers, there can be (and are) many different varieties of DAP clients and servers. The only thing that ties them together is that the client making the request and the server fulfilling it follow the strictures of this specification.

The DAP specification describes the dialog between requesting clients and responding servers, but it does not specify the implementation of that dialog. So long as your program can hold up its end of the conversation, there is no limit on how it is done. Specifically, this means that though the DAP specifies the persistent representation of its abstract data types (this is the form taken by a message), it does not specify the data structures that may implement these data types in a computer program. This also means that the programs can use any means to transport the requests and responses, although the protocol has been written with protocols such as SOAP over HTTP, GridFTP, *et c.*, in mind.

1.1 Lexicographical Conventions

The DAP data model contains a model element called an *Attribute*. XML element tags also contain attributes. In order to avoid confusion in this document when we are referring to OPeNDAP *Attributes* we will capitalize the first letter (“A”). When we are referring to XML attributes we will not capitalize the first letter (“a”). In the event that this distinction is not adequate we will endeavor to make the distinction clear, either from context or from additional illuminating language.

[1] Note that OPeNDAP refers to a project, managed by OPeNDAP, Inc., a Rhode Island not-for-profit corporation, while DAP refers to the Data Access Protocol which is a central component of the OPeNDAP project.

2 Variables

The DAP characterizes a data source as a collection of variables. Each variable consists of a name, a type, a value, and a collection of *Attributes*. *Attributes*, in turn, are themselves composed of a name, a type, and a value (Section 3 on page 11). The distinction between information in a variable and in an *Attribute* is somewhat arbitrary, especially in the case of global *Attributes*. However, the intention is that *Attributes* hold information that aids in the interpretation of data held in a variable.[2] Variables, on the other hand, hold the primary content of a data source.

Each variable in a data source MUST have a name, a type and some value or values. Using just this information and armed with an understanding of the definition of the DAP data types, a program can read any or all of the information from a data source. The names and types of a data source's variables comprise its *syntactic metadata* .[10]

The DAP variables come in several different types. There are several *atomic* types, the basic indivisible types representing integers, floating point numbers and the like, and four *constructor* types (also called *container* types) which can be used to define new types by combining instances of both the atomic and constructor types.

This section describes the abstractions that constitute the variable type menagerie: the range of values and the kind of data each type can represent. For each abstract variable type, there is a more concrete persistent representation, which is the information actually communicated between a DAP server and its clients. The persistent representation consists of two parts: the declaration of the type and the encoding of its value(s). For a description of the persistent representation see Section 6 (page 18) . To see how the types are to be declared, see Section 6.1 (page 18) . For the encoding of these variable types (how they are to be packaged for transmission) see Section 6.5 (page 36) .

Each variable MAY have one or more *Attributes* associated with it. For information about *Attributes*, see Section 3 (page 11) .

2.1 Atomic variables

As their name suggests, *atomic* data types are indivisible. There are no constraint expression operators that can be used to request part of an instance of one of these types (Section 4 on page 12). These variables are used to store integers, enumerations, booleans and real numbers as well as strings, URLs and times. There are four families of atomic types, with each family containing one or more variation:

- Integer, Boolean and Enumeration types
- Floating-point types
- String types
- Binary images

2.1.1 Integer types

The integer types are summarized in Table 1. Each of the types is loosely based on the corresponding data type in ANSI C [13]. However, the DAP, unlike ANSI C, does specify the bit-size of each of the integer types. This is done so that when values are transfered between machines they will be held in the same type of variable, at least within the limits of the software that implements the DAP.

[2] *Attributes* appear in many data storage systems such as netCDF[9], HDF4[7] and HDF5[8]. They also appear under the moniker 'property' in Common Lisp[11].

NOTE: *move this to a more appropriate section, maybe its own... Maybe a section that combines the length spec and other stuff that is used in several places. jhrg 11/2/03*

When implementing the DAP, it is important, of course, to match information in a data source or read from a DAP response to the local data type which best fits those data. In some cases and exact match may not be possible. For example Java lacks unsigned integer types[2]. Implementation faced with such limitations MUST ensure that clients will be able to retrieve the full range of values from the data source. As a practical consideration, this may be implemented by hiding the variable in question or returning an error.

If a variable is automatically hidden (*i.e.*, the server analyzes the data source and determines that a particular variable cannot be represented correctly and automatically removes it from those variables made accessible using the DAP, this MUST be noted by adding a global *Attribute* to the data source indicating this has taken place. The note MUST include the name of the variable(s) and the reason(s) for their exclusion. If a variable is removed by a human, this *Attribute* is OPTIONAL.

In their persistent representation in the DAP, integer values are stored as *twos-compliment big-endian* numbers.[3] See Section 6.5.3 (page 37) .

Table 1: The DAP Integer Data types.

name	description	range
Byte	8-bit unsigned char	0 to $2^8 - 1$
Int16	16-bit signed short integer	-2^{15} to $2^{15} - 1$
Uint16	16-bit unsigned short integer	0 to 2^{16}
Int32	32-bit signed integer	-2^{31} to $2^{31} - 1$
Uint32	32-bit unsigned integer	0 to 2^{32}
Int64	64-bit signed integer	-2^{63} to $2^{63} - 1$
Uint64	64-bit unsigned integer	0 to 2^{64}

2.1.2 Booleans

Data which can take on only one of two values, true or false, may be represented using the *Boolean* data type. This type is used by data storage software such as HDF5[8] and data communication specifications such as ASN.1[5].

See Section 6.5.3 (page 37) for the description of how *Booleans* are encoded for transmission.

Table 2: The DAP *Boolean* type.

name	description
Boolean	One of two possible values: either true or false.

2.1.3 Enumerations

An *Enumeration* is used to represent a set of discrete named values. The values MUST be integers between -2^{31} and $2^{31} - 1$. No value may be used more than once. The intent is that the size of the set will be small; an *Enumeration* should not be used to represent a set of thousands or millions of values, although there's nothing in principal preventing such a use. To represent the values, a signed 32-bit integer is used. An *Enumeration* MUST include a symbolic name for each integer value.

See Section 6.5.3 (page 37) for the description of how *Enumerations* are encoded for transmission.

[3] Big-endian is the default byte order for data transmissions, but see Section 5.1 (page 17) regarding negotiation of byte order.

Table 3: The DAP *Enumeration* type.

name	description
Enumeration	a set of unique discrete integral values greater than or equal to zero, each enumerated and bound to symbol.

2.1.4 Floating point types

The floating point data types are summarized in Table 4. The two floating point data types use IEEE 754 [15] to represent values. The two types correspond to ANSI C's `float` and `double` data types.

In their persistent representation, floating point values are stored by default using big-endian notation. See Section 6.5.3 (page 37) .

Table 4: The DAP Floating Point Data types.

name	description	range
Float32	IEEE 32-bit floating point [15]	$\pm 1.175494351 \times 10^{-38}$ to $\pm 3.402823466 \times 10^{38}$
Float64	IEEE 64-bit floating point	$\pm 2.2250738585072014 \times 10^{-308}$ to $\pm 1.7976931348623157 \times 10^{308}$

2.1.5 String types

The string data types are summarized in Table 5. There are three. The first is a simple string type corresponding to the ANSI C notion of a string: a series of Unicode (ISO 10646) characters. The DAP uses the UTF-8 encoding of Unicode characters.

There is no limit to the size of a *String*; the length is specified using a *Length Specifiaction* (See Section 6.5.1 on page 37). Unicode characters can each be several bytes long, but note that UTF-8 encoding is identical to US-ASCII encoding for character values up to 127 (hexadecimal `7f`). This means that strings that contain only characters from the 7-bit ASCII set are one byte per character, and use the standard ASCII encoding. Characters from 128 to 255 (hex `80` to `ff`) are encoded into two bytes in UTF-8 [1].

The DAP also provides a *URL* data type which is identical to a *String*, but has the specific meaning of a pointer to some WWW resource. In DAP applications, this is usually used to refer to another data source, in a manner reminiscent of a C pointer. Unlike the *String* type, a *URL* is limited to standard (7-bit) US-ASCII characters, due to the limitations of the syntax of Internet URLs[3].

The last string type the DAP provides is the *Time* data type which is identical to a *String*, but has the specific meaning of an ISO8601[6] date/time string. The ISO Date/Time standard provides a way to encode dates, both local and UTC times and time ranges[12]. The *Time* data type is included in the DAP so that date and time information may be represented in a standard fashion.

In general, most data sources will not use ISO8601 date-time strings; servers SHOULD provide both the native representation of date-time information and the ISO8601 representation. This will allow savvy clients to exploit the native representation while more generic clients can access the data source without the need to accommodate its quirks.

Strings are individually sized. This means that in constructor data types containing multiple instances of some *String*, such as *Sequences* and *Arrays*, successive instances of that *String* MAY be of different sizes.

See Section 6.5.3 (page 37) for other details of the persistent representation of *Strings*.

2.1.6 Binary images

Binary Images are uninterpreted, opaque, lumps of digital data. There is no limit to the size of a *Binary Image*; the length is specified using a *Length Specifiaction* (See Section 6.5.1 on page 37). They are meant as a way for a server

Table 5: The DAP String Data types.

name	description
String	a series of Unicode (UTF-8) characters.
URL	a series of US-ASCII characters (the Internet doesn't support Unicode in URLs), meant to represent an on-line resource somewhere, usually another data source.
Time	a series of Unicode (UTF-8) characters which contain a valid ISO 8601 date/time string.[6]

to pass elaborate data types to a client without having to encode them in the DAP data model. For example, a digital sound clip, say an MP3 file, could be represented as a one-dimensional DAP *Array* of integer values. But if the server stores these as MP3 files, and the client can play them as such, then it may not be efficient to convert from MP3 to the DAP *Array* and then back again.[4]

Binary Images are individually sized. This means that in constructor data types containing multiple instances of some image, such as *Sequences* and *Arrays*, successive instances of that *Binary Image* MAY be of different sizes.

See Section 6.5.3 (page 37) for the description of how binary images are encoded for transmission.

2.2 Constructor variables

The *constructor* types are assembled from collections of other variables. A constructor type MAY contain both atomic and constructor types. There are no restrictions on the number of levels of nesting.

There are four constructor data types:

- *Array*
- *Structure*
- *Grid*
- *Sequence*

2.2.1 *Array*

An *Array* is a one-dimensional indexed data structure similar to that defined by ANSI C. An *Array*'s member variable may be of any DAP data type with the exception of the *Enumeration* type.

Multidimensional *Arrays* are defined as *Arrays* of *Arrays*. Multi-dimensional *Arrays* are stored in *row-major* order (as is the case with ANSI C). The size of each *Array*'s dimensions MUST be given. There is no limit to the size or number of an *Array*'s dimensions; the length is specified using a *Length Specifiaction* (See Section 6.5.1 on page 37).

Each dimension of an *Array* MAY also be named.

Arrays of *Strings* and binary images MAY contain elements of varying lengths. However, multi-dimensional *Arrays* MAY NOT have rows or columns of varying lengths. If you need a structure like this, consider a *Sequence* of *Sequences* or a *Sequence* of *Arrays*. Note that in the latter case, a *Sequence* of *Arrays*, each instance of the array MUST be the same size.[5]

See Section 2.2.4 (page 9) for more about the possibilities and the limitations.

Note: *Arrays* of *Enumerations* are NOT allowed.

[4]Note that, as with the case of the MP3, the word "images" does not necessarily refer to image data, though obviously you can use a *Binary Image* to transmit GIF data, for example. The word only implies that the binary data within a *Binary Image* is uninterpreted, and MUST be preserved intact through any representation transformation.

[5]The types of the variables which comprise a Sequence are be the same for each instance (*i.e.* 'row') of that Sequence. For arrays, the 'shape' is part of the type.Where the protocol demands that Sequence be used, an interface for client application programs is free to make those look like arrays with varying lengths.

2.2.2 *Structure*

A *Structure* groups variables so that the collection can be manipulated as a single item. The *Structure*'s member variables MAY be of any type, including other constructor types. The order of items in the *Structure* is significant only in relation to the persistent representation of that *Structure*.

There is a special case of the *Structure* data type, called *Dataset*. This is the container that encompasses all the variables provided in some data source.

2.2.3 *Grid*

A *Grid* is a special case of a *Structure*, used to supply information to aid in the interpretation of *Arrays*. A *Grid* sets up an association between a target *Array* and a collection of "map" *Arrays*. Each dimension of the target array MUST correspond to one or more dimensions of the map arrays. For example, a two-dimensional target array could map to a collection of identically-sized two-dimensional map arrays, or to an assortment of one-dimensional map vectors. A three dimensional array might map to a collection of one-, two-, and three-dimensional map arrays.

A common use for this kind of data might be raw satellite data, where measurements are frequently not on a regular latitude-longitude grid. In the example shown, data values (z_{mn}) can be associated with arbitrary latitude (y_{mn}) and longitude (x_{mn}) values, while still retaining their gridded nature. One can easily add a time dimension to the collection, as well. The result is that any element in the target array has a corresponding latitude, longitude, and time value.

$$target = \begin{bmatrix} z_{11} & z_{21} & \cdots & z_{m1} \\ z_{12} & z_{22} & \cdots & z_{m2} \\ z_{13} & z_{23} & \cdots & z_{m3} \\ \vdots & \vdots & \ddots & \vdots \\ z_{1n} & z_{2n} & \cdots & z_{mn} \end{bmatrix}$$

$$map_x = \begin{bmatrix} x_{11} & x_{21} & \cdots & x_{m1} \\ x_{12} & x_{22} & \cdots & x_{m2} \\ x_{13} & x_{23} & \cdots & x_{m3} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1n} & x_{2n} & \cdots & x_{mn} \end{bmatrix} map_y = \begin{bmatrix} y_{11} & y_{21} & \cdots & y_{m1} \\ y_{12} & y_{22} & \cdots & y_{m2} \\ y_{13} & y_{23} & \cdots & y_{m3} \\ \vdots & \vdots & \ddots & \vdots \\ y_{1n} & y_{2n} & \cdots & y_{mn} \end{bmatrix} map_t = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_n \end{bmatrix}$$

This *Grid* indicates that each z value z_{ij} corresponds to the values of the x and y maps at (i, j). It also indicates that each row of z_i (and x_i and y_i) correspond to an element of the column vector t. Such a data structure might be used to hold satellite data before it has been processed into a 'picture.' In that case the z target array might be a reflectance value from the satellite's sensor, the x and y maps would provide the latitude and longitude for each pixel in z and the t map would hold the time at which each scan line was collected.

A *Grid* will always contain a target array, and at least one map array. The only other requirement is that each dimension of the target array MUST correspond to one or more dimensions in the map arrays. The arrays in the *Grid*—target or map— MAY be an *Alias* to arrays somewhere else in the dataset (Section 2.4 on page 10). This can save transmission bandwidth by avoiding the repetition of data when maps are common to more than one *Grid*.

A special case of *Grid* is an association of an N dimensional *Array* with N vectors (one-dimensional *map vectors*), each of which has the same number of elements as the corresponding dimension of the *Array*. Each vector is used to map indexes of one of the *Array*'s dimensions to a set of values which are normally non-integer (*e.g.*, floating point values).

Schematically, the special case of the *Grid* is like the following:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_m \\ z_{11} & z_{21} & z_{31} & \cdots & z_{m1} \\ z_{12} & z_{22} & z_{32} & \cdots & z_{m2} \\ z_{13} & z_{23} & z_{33} & \cdots & z_{m3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ z_{1n} & z_{2n} & z_{3n} & \cdots & z_{mn} \end{bmatrix}$$

Each column of the z *Array* corresponds to an entry in the x map vector, and each row of z corresponds to some y value. So, for example, the data value at $z_{42,33}$ corresponds to the values x_{42} and y_{33}.

The *Grid* type was created to deal with geo-located data, with irregular spacing of the rows and columns, which is useful when converting to and from different map projections. But the *Grid* structure is more generally useful. For example, one of the map vectors could be an *Array* of (x, y) pairs (stored in a *Structure*), and the other a series of time values, and the *Grid* would become a record of several synoptic time series. The maps MUST be *Arrays*, but the *Arrays* MAY be collections of any DAP data type except *Sequence*, *Grid*, *Binary Image*, or *Enumeration*.[6]

2.2.4 *Sequence*

A *Sequence* can best be described as an ordered collection of zero or more *Structures*. Each instance in the series consists of the same set of variables, but contains different values.

The semantics of the *Sequence* data type are very close to those of a table in a relational database. You can think of the instances in a *Sequence* as rows in a traditional relational table. OPeNDAP servers that serve data from a DBMS like Oracle or mySQL use *Sequences* to reflect the structure of their data.

A *Sequence* S can be represented as:

$$\begin{array}{cccc} s_{11} & s_{21} & \cdots & s_{n1} \\ s_{12} & s_{22} & \cdots & s_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ s_{1i} & s_{2i} & \cdots & s_{ni} \\ \vdots & \vdots & \vdots & \vdots \end{array}$$

Where each $s_1 \cdots s_n$ entry represents a set of DAP variables, and the collection of such entries constitutes the *Sequence*. Every entry of *Sequence* S has the same number, order, and type of variables. If s_{21} is a *Float64*, then all the s_{2i} will also be *Float64* variables. Similarly, in a *Sequence* which contains an *Array* or *Structure*, each instance of the *Array* or *Structure* will be the same size. However, a *Sequence* MAY contain a *Sequence* and each instance of the interior *Sequence* MAY have a different number of entries. Also, unlike an *Array*, a *Sequence* has no explicit size.

> Note that though the semantics of *Sequences* places limitations on the kinds of requests a client may make of a server, once the *Sequence* has been retrieved, a client program may reference it in any way desired. The DAP defines the persistent representation of data types, and the interaction between client and server (which includes what kinds of requests can be made for what kind of variables), but the DAP does not specify the internal implementation of the data types for any client or server.

2.3 Names

A DAP variable's name MAY contain any UTF-8 characters. However, some interfaces may require that any characters not part of US-ASCII be escaped so that the names are represented in US-ASCII.[7]

2.3.1 Constructor variable names

The members of a constructor variable can be individually addressed in the following fashion:

Array Individual items are addressed with a subscript. For an *Array* named `Temp`, the fourteenth member of the *Array* is referenced as `Temp[13]` (all indexes start at zero). A two-dimensional *Array* is addressed with two subscripts, contained in separate brackets: `SurfaceTemp[13][3]`. See Section 4 (page 12) .

Structure Members of the *Structure* are addressed by appending the member name to the *Structure* name, separated by a forward slash (`/`). If the *Structure* `Position` has a member named `Height`, then it is addressed as `Position/Height`. The members of a *Structure* MUST have different names from one another.

[6] This restriction has been put in place to keep writing general clients tractable. If the set of data types in a *Grid*'s map *Arrays* is allowed to be a *Sequence*, for example, any general client would have to be capable of processing that data type in a response. Such a client would be very hard to build.

[7] The HTTP GET interface from OPeNDAP will require this if it's implemented for DAP 4.

Grid The arrays in a *Grid* MAY be referenced in the same fashion as the members of a *Structure*. For a two-dimensional *Grid* named `Cloud`, with one-dimensional map vectors `Latitude` and `Longitude`, a member of a map vector is be addressed like this: `Cloud/Latitude[36]`. This refers to a single latitude value. You can also request part of the target array: `Cloud/Cloud[36][42]`, which will return a single data measurement. The *Grid* itself MAY be addressed like an *Array*: `Cloud[36][42]`, which will return the same value as `Cloud/Cloud[36][42]`, but as a *Grid*. See Section 4.2 (page 16) for an explanation of how data types are transformed by constraints.

Sequence A *Sequence* member is addressed in the same fashion as a *Structure*. That is, a time called `Releasedate` of a *Sequence* named `Balloons` is addressed as `Balloons/Releasedate`. But note that unlike a *Structure*, this name references as many different values as there are entries in the `Balloons` *Sequence*. A single entry or range of entries in a *Sequence* MAY be addressed with a hyperslab operator like the items in an *Array*. The variables in a *Sequence* MUST have different names from one another.

2.3.2 Fully Qualified Names

Variables and *Attributes* exist in the same name-space. This fact has significant impacts on the way that datasets can be organized. There cannot exist a variable and an *Attribute* of the same name at the same level of a dataset. This is primarily of concern for constructor variables: If a constructor variable has a member variable named *time* then it MAY NOT also have an *Attribute* named *time*. This rule allows the *fully qualified name* of each *Attribute* and variable in a *Dataset* to be unique. Note that in this example, the variable *time* is essentially an *Attribute* structure inside of the parent constructor variable. In essence all variable members variables of a constructor variable act as *Attribute* structures within the constructor variable.

Variable Names

The *fully qualified name* of a variable is composed of the ordered collection of variable names, starting at the *Dataset* level but not including the *Dataset* name, that can be followed to the terminal variable name. The names MUST be separated by the slash ("/") character, and the *fully qualified name* MUST begin with the slash ("/") character. Said another way, the *fully qualified name* of any variable in a *Dataset* is the concatenation of the variable's name, preceded by the forward slash separated list of the names of the Constructor variables that contain it. The first name MUST be a variable name at the *Dataset* level preceded by a forward slash.

Thus, if a *Dataset* named `test` contains a structure named `sst` which contains a variable named `foo`, the *fully qualified name* would be `/sst/foo`.

Attribute Names

The *fully qualified name* of an *Attribute* is composed of the ordered collection of variable and *Attribute* names, starting at the *Dataset* level but not including the *Dataset* name, that can be followed to the terminal source *Attribute*. The names MUST be separated by the slash ("/") character, and the *fully qualified name* MUST begin with the slash ("/") character. If a *fully qualified name* for an *Attribute* terminates with a regular variable name, then it will be interpreted to refer to the collection of *Attributes* associated with said variable.

Thus, if a *Dataset* named `test` contains a structure named `sst` which contains a variable named `foo`, the *fully qualified name* of the *Attributes* of foo would be `/sst/foo`. If `foo` possessed an *Attribute* named `fruit` then the *fully qualified name* for `fruit` would be `/sst/foo/fruit`.

2.4 Variable Aliases

A variable in a DAP data source might contain no data of its own, but simply be a pointer to some other variable in the set. Such a variable is called an *Alias*. An *Alias* is useful for achieving compatibility with other data sources, for conforming to metadata requirements, and for conserving bandwidth in large data transmissions.

A variable *Alias* MAY refer only to another variable. It MAY NOT refer to an *Attribute*. It MAY NOT refer to another variable *Alias*.

An *Alias* MUST be defined using the *fully qualified name* (see Section 2.3.2 on page 10) of the variable to which it refers. This means that if a *Dataset* called `test` contains a *Structure* called `S` that contains both a variable called `V` and a variable *Alias* directed at that variable called `M`, then `M` MUST reference `V` as `/test/S/V`.

There are significant restrictions on the use of *Aliases* in conjunction with the *Sequence* data-type. An *Alias* member of a *Sequence* MAY refer to a member variable of the same *Sequence*. An *Alias* member of a *Sequence* MAY NOT

refer to a variable outside of the *Sequence*. This may seem a little arbitrary at first, but consider that every member variable of a *Sequence* is multi-valued. It makes little sense to allow an *Alias* to refer to such a variable, unless the *Alias* exists in the same dimensional space. This can only be guaranteed if the *Alias* is a member of the same *Sequence* as the variable to which it refers.

See the discussion of the *Alias* element in Section 6.3.1 (page 20) for specific information about the syntax of an *Alias*.

3 Attributes

Each variable in a data source MAY have zero or more *Attributes* associated with it. The entire dataset (see Section 2.2.2 on page 8) MAY itself have *Attributes*, too. These are called *global* Attributes . All *Attributes* are held within *Attribute structures* , even when there is only one *Attribute* associated with a variable. Every variable acts as an *Attribute* structure. This includes the *Dataset* type, which contains the global *Attributes*.

While the DAP does not require any particular *Attributes*, some may be required by various *metadata conventions* . The *Attributes* associated with a data source and its variables comprise the *semantic metadata* for that data source.

The data model for *Attributes* is somewhat simpler than that for variables. An *Attribute* MAY be be set of 1 or more `values` of the same *atomic type*, OR it MAY be a `structure` that contains other *Attributes*, but not individual `values`. In other words an *Attribute* MAY be a list of values of the same atomic type, or it MAY be a container of additional *Attributes*. An *Attribute* MAY NOT contain both `values` and other *Attributes*.

An *Attribute*'s value MAY be a any of the following atomic types:

- `Boolean`
- `Byte`
- `Int16`
- `UInt16`
- `Int32`
- `UInt32`
- `Int64`
- `UInt64`
- `Float32`
- `Float64`
- `String`
- `URL`

An *Attribute* that contains other *Attributes* MUST be of type `structure`.

There are two special types of *Attributes*: `modifier` and `alias`. *Attributes* of type `alias` are discussed in Section 3.1 (page 11) . *Attributes* of type `modifier` are discussed in Section 3.2 (page 12) .

There are examples of *Attribute* definitions in the description of the *Attribute* element in Section 6.3.3 (page 21) .

3.1 Attribute Aliases

A special type of *Attribute* is *alias* . In a manner comparable to a variable *Alias* (see Section 2.4 on page 10), a variable's *Attributes* MAY also be aliased, with an *Attribute* of one name referring to a different *Attribute* (possibly with a different name). An *Attribute* of one variable MAY be an alias that refers to an *Attribute* of another variable. *Attribute* aliases MAY only refer to *Attributes*. They MAY NOT refer to variables or other *Attribute* aliases.

An *Attribute* alias MUST be defined using the *fully qualified name* of the *Attribute* to which it refers. See Section 2.3.2 (page 10)

See the discussion of the *Attribute* `alias` type in Section 4 (page 21) for specific information about the syntax of an *Attribute* `alias`.

3.2 Processing Attributes

The processing *Attributes* are a special set of *Attributes* used to record modifications to a dataset's *Attributes* or its data. After a dataset is released, other people may copy the data and make it available in modified form, or with secondary data products added, or with *Attribute* information that may make the data conform to some data standard. However useful these modifications might be, it is essential that users of some dataset be able to determine which parts of that set are original, and which have been added subsequent to the original publication. The processing *Attributes* exist to provide a way to create an "audit trail" that will permit users to determine how a dataset has been modified.

The key piece of information is a global *Attribute* of type `modifier`, which is essentially the signature of the organization that has performed the modification. A `modifier` *Attribute* contains:

- A DAP URL for the original source of data. This MUST be held in a *URL Attribute* and the name of the *Attribute* MUST be `origin_server`;
- The name or description of the organization that is providing the modified data. This MUST be held in a *String Attribute* and the name of the *Attribute* MUST be `organization`;
- The URL of the service which has introduced the changes. This MUST be held in a *URL Attribute* and MUST be named `modifying_service`.

Attributes of type `modifier` MAY only appear at the top level of a data set. They MAY not be used as members of other *Attributes* or of other variable *Attribute* structures. Example 5 on page 23 shows an example of a modifier *Attribute*.

4 Constraint Expressions

A *constraint expression* provides a way for DAP client programs to request certain variables, or parts of certain variables, from a dataset. Many datasets are large and many variables in datasets are also large. Clients are often interested in only a small number of values from the entire dataset. Constraint expressions provide a way for clients to tell a server which variables, and in many cases, which parts of those variables, they would like. In addition, the constraint expression can be used to request that the server to omit the *Attribute* information from a DDX.

This section presents the subsamping abilities that MUST be provided by a DAP compliant server. It does so without binding these capabilities to any particular syntax; see Section 7 (page 41) for the XML representation of a constraint expression. Some transport protocols may choose to implement additional syntaxes but MUST implement the syntax described in Section 7.

Note that an empty constraint expression implies that the entire data source is to be accessed.

4.1 Limiting data by type and by value

A constraint expression provides two different methods to access the information held by a data source. The constraint expression can be used to limit data using the names of variables or by scanning variables and returning only those values that satisfy certain relational expressions. The former are reffered to a *projections* while the latter are called *selections* .

A constraint expression MAY combine both projection and selection constraints. For example, a projection might specify that temperatures held in a *Sequence* are to be returned, and a selection would specify that only *Sequence* entries with dates later than 1999 are to be examined. The result returned from a request like this would be a *Sequence* of temperature measurements taken after 1999.

Section 4.1.1 (page 13) describes the projection operations which any DAP implementation MUST support and, likewise, Section 4.1.2 (page 14) describes the required selection operations.

To provide implementors with a means to extend the constraint expression mechanism, it is possible to add functions to a server and to call those as part of the constraint expression. Functions are described in Section 4.1.3 (page 16) .

4.1.1 Projections

The *projection clause* of a constraint expression provides a way to choose parts of a data set based on the shape of the *Dataset* and the variables that comprise it. There are two types of projection operations. First, it is possible to choose individual fields of the constructor data types. This is called *field projection* and applies to the *Structure*, *Grid* and *Sequence* data types in the following ways:

Structure A field projection which chooses one or more fields from a *Structure* variable causes a DAP server to return only those named fields from the *Structure*. Note that the *Dataset* itself is a *Structure*.

Grid A field projection which chooses one or more fields from a *Grid* variable causes a DAP server to return only those named fields from the *Grid*. It is likely that the variable returned will no longer meet the criteria for a correctly formed *Grid* data type, so the variable may be returned as a *Structure* instead (see Section 4.2 on page 16).

Sequence A field projection which chooses one or more fields from a *Sequence* variable causes a DAP server to return only those named fields from the *Sequence*. For the *Sequence* type, this means returning the N instances but limiting the fields those given the in the field projection. For example, suppose the *Sequence* S has 3 fields:

$$\begin{array}{ccccc} s_{11} & s_{21} & s_{31} & \cdots & s_{p1} \\ s_{12} & s_{22} & s_{32} & \cdots & s_{p1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ s_{1i} & s_{2i} & s_{3i} & \cdots & s_{pi} \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{array}$$

If a field projection is used to choose only the second field, the result of accessing S would be:

$$\begin{array}{c} s_{21} \\ s_{22} \\ s_{23} \\ s_{24} \\ \vdots \end{array}$$

The second type of projection is a *hyperslab* . A hyperslab is used to limit returned data to those elements that fall within a range of index values, and MAY also specify that the range be subsampled using a *stride* . By including a hyperslab projection for one or more dimensions of a variable it is implied that any unnamed dimensions are to be returned in their entirety.[8] A hyperslab is applied to the *Array*, *Grid* and *Sequence* types in the following way:

Array *Array* dimensions are numbered $0, \ldots, N-1$ for an *Array* of rank N. Within each dimension of size M, elements are numbered $0, \ldots, M-1$. A hyperslab projection for dimension $n, 0 \leq n < N$ MUST include the starting index i_{n_s} and ending index i_{n_e} such that $i_{n_s} \leq i_{n_e} \forall \{0 \leq i_n < M\}$. If a stride is included in the hyperslab and is greater than $i_{n_e} - i_{n_s}$ then the hyperslab is equivalent to one where $i_{n_s} = i_{n_e}$ and the original value of i_{n_e} is discarded.

Grid *Grid* dimensions are numbered as are *Array* dimensions; *Grid* dimensions MAY have hyperslab projetions applied to them in a manner similar to *Arrays* except that a hyperslab applied to a *Grid* is applied to not only the target array, but also all the corresponding map arrays. For example, given the *Grid*:

$$target = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} map_1 = \begin{bmatrix} -53 & -52 & -51 & -50 \\ -52 & -51 & -50 & -49 \\ -51 & -50 & -49 & -48 \\ -50 & -49 & -48 & -47 \end{bmatrix} map_2 = \begin{bmatrix} 26 & 25 & 24 & 23 \\ 25 & 24 & 23 & 22 \\ 24 & 23 & 22 & 21 \\ 23 & 22 & 21 & 20 \end{bmatrix}$$

[8] For some interfaces, it may be necessary to place more restrictions on hyperslab projections.

A hyperslab projection which chose row indexes 1 and 2 and column indexes 1 and 2 would cause a server to return:

$$target = \begin{bmatrix} 6 & 7 \\ 10 & 11 \end{bmatrix} map_1 = \begin{bmatrix} -51 & -50 \\ -50 & -49 \end{bmatrix} map_2 = \begin{bmatrix} 24 & 23 \\ 23 & 22 \end{bmatrix}$$

for the *Grid*.

Note that a field and hyperslab projection can be combined for a *Grid* to choose only part of one of the fields, say just part of the the target *Array*. In this case, the hyperslab applied to one field of the *Grid* is equivalent to a hyperslab applied to an *Array*. Effectively, the field projection yields an *Array* and the hyperslab is then applied to that *Array*.

Sequence A hyperslab can be applied to a *Sequence*. A *Sequence* with M instances can have a hyperslab projection applied to it as if it is an *Array* of rank 1. Since the *Sequence* type does not contain an explicit dimension size, the size M is not known until the entire *Sequence* is accessed.[9] A hyperslab projection can be used to ask for the first m elements, the next m elements, etc., which may be very useful for clients which need to know the sizes of varaibles before accessing them. A hyperslab projection for a *Sequence* (i_s, i_e) will return m instances of the *Sequence* such that $m = \lfloor i_e, M - 1 \rfloor - i_s$ depending on whether i_e is an index greater than the number of instances in the *Sequence*.

It is possible to ask for values from several variables in a single constraint expression by including several projections in the constraint expression. Also note that an empty constraint expression, by convention, projects all of every variable in a data source.

4.1.2 Selections

A *selection* provides a way to limit data accessed based on the value(s) of those data. In many ways selections are similar to WHERE claues in SQL[4]. A selection is comprised of one or more relational sub-expressions. Each sub-expression MUST be bound to a variable listed in a projection clause. When several sub-expressions comprise a selection, the boolean value of the selection is the logical `AND` of each of the boolean values of each sub-expression. Note that there is no way to perform a logical `OR` operation on the sub-expressions but there is a way, within a sub-expression, to test several values and return `true` if any satisfy the releation.

Each of the relational sub-expressions (*i.e.*, relations) is composed of two operands and a relational operator. Each operand MUST be an atomic data type; it MAY be a *fully qualified name* from the data source or a constant. In some cases there are further limitations on the allowed types based on the relational operator. Table 6 lists the operators, their meaning and the data types on which they may be applied.

The operands in a relation MAY be either single or multi-valued. If an operand has more than one value, each value is used in succession when evaluating the relation. For example, suppose there is a relation:

$$site = \{\text{“}Diamond\_St\text{”}, \text{“}Blacktail\_Loop\text{”}\}$$

Then that relation is true for any instance where $site$ is either “Diamond_St” `OR` “Blacktail_Loop”.

Selections MAY be applied to *Sequence* and *Grid* data types in the following ways:

Sequence Logically, the relations in a selction bound to a *Sequence* are evaluated once for every instance (*i.e.*, row) of the *Sequence*; the result of applying the selection to the *Sequence* is a *Sequence* where all of the instances satisfy all of the relations.

A *Sequence* S with three fields and four instances such as:

$index$	$temperature$	$site$
10	17.2	$Diamond\_St$
11	15.1	$Blacktail\_Loop$
12	15.3	$Platium\_St$
13	15.1	$Kodiak\_Trail$

[9] For many *Sequence* variables, it may never be the case that the entire *Sequence* is accessed since it may contain millons of instnaces.

Table 6: DAP Selection Relational Operators

Operator	Meaning	Types
$<$	Less than	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Enumeration, Float32, Float64, Time
$\leq$	Less than or equal to	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Enumeration, Float32, Float64, Time
$>$	Greater than	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Enumeration, Float32, Float64, Time
$\geq$	Greater than or equal to	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Enumeration, Float32, Float64, Time
$=$	Equal	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Boolean, Enumeration, Float32, Float64, String, Url, Time
$\neq$	Not equal	Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Boolean, Enumeration, Float32, Float64, String, Url, Time
`=~`	Regular expression match	String, Url, Time

A selection such as $index \geq 11$ would choose the last three instances:

$index$	$temperature$	$site$
11	15.1	$Blacktail\_Loop$
12	15.3	$Platium\_St$
13	15.1	$Kodiak\_Trail$

The selection $site$`=~`"$.*\_St$" would choose two instances:

$index$	$temperature$	$site$
10	17.2	$Diamond\_St$
12	15.3	$Platium\_St$

And a selection with the two sub-expressions $index \leq 11$, $site$`=~`"$.*\_St$" would return only one instance:

$index$	$temperature$	$site$
10	17.2	$Diamond\_St$

Grid When selections are applied to *Grids* with multi-dimensional map arrays, the returned data will be the smallest rectangular (contiguous) subset of the *Grid* that contains all the data that satisfies the constraint. For example, suppose there is a *Grid* which has a target array of rank two and two map arrays, each of which also have rank two, such as:

$$target = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} lat = \begin{bmatrix} -53 & -52 & -51 & -50 \\ -52 & -51 & -50 & -49 \\ -51 & -50 & -49 & -48 \\ -50 & -49 & -48 & -47 \end{bmatrix} lon = \begin{bmatrix} 26 & 25 & 24 & 23 \\ 25 & 24 & 23 & 22 \\ 24 & 23 & 22 & 21 \\ 23 & 22 & 21 & 20 \end{bmatrix}$$

Suppose the *Grid* is constrained by a selection clause that limits returned values to those where `lat` is greater than 24.5 and `lon` is less than -50.5. The *Grid* returned by such a constraint would be:

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} -53 & -52 \\ -52 & -51 \end{bmatrix} \begin{bmatrix} 26 & 25 \\ 25 & 24 \end{bmatrix}$$

Notice that one point is returned that does not satisfy the *Select* provision. *Grids* are constrained to be rectangular.

4.1.3 Server Functions

A constraint expression can also use functions executed by the server. These can appear in a selection or in a projection, although there are restrictions about the data types functions can return.

A function which appears in the projection clause MAY return any of the DAP data types. In this case the return value of the function is treated as if it is a variable present in the top level of the *Dataset*.

A function which appears in the selection clause MAY return any atomic type if it is used as an argument in one of the relational sub-expressions. If a function in the selection clause is used as the entire sub-expression, it MUST return a *Boolean* value.

When functions encounter an error, a DAP server MUST signal that condition by returning an error response. A server MAY NOT return a partial response; any error encountered while evaluating the constraint expression MUST result in a response that contains an unambiguous error message.

4.2 Data Type Transformation Through Constraints

When a constraint expression has a projection clause that identifies a piece of a constructor variable, such as one field of a *Structure* or just the array part of a *Grid*, the *lexical scoping* of the variable is not abandoned. This is important for avoiding name collisions. For example, if you request only one item from a *Structure*, you get a *Structure* returned that has only one member variable.

Here is the behavior for each data type:

Array An *Array* is always returned as an *Array* of the same rank as the source *Array*. A hyperslab request that effectively eliminates a dimension by reducing its size to 1 does *not* reduce the rank of the returned *Array*. For example, suppose a 10 by 10 element *Array* was subsampled to a 1 by 2 *Array*. The returned variable would still be described as a two dimensional *Array*.

Structure A *Structure* is always returned as a *Structure*. If the projection clause of a constraint expression selects only one member of the *Structure*, then a one-member *Structure* is returned. If more than one member of the *Structure* are named in the projection clause, they MUST be returned in the same *Structure*.

Grid A *Grid* modified with a hyperslab operator will return another *Grid*, following the same rules as an *Array*. But if the projection clause specifies the elements of the *Grid* independently of one another—the target array, or one of the maps—then a *Structure* is returned containing only the specified variables. A two-dimensional *Grid* named `Cloud` will return a *Grid* in response to a request like this: `Cloud[1:10][20:30]`. But a request for the target array alone—`Cloud/Cloud[1:10][20:30]`— will return a *Structure* called `Cloud` containing an *Array* called `Cloud`. The map arrays will not be returned.

A *Grid* modified with a selection will remain a *Grid*. The return value of such a constraint is the smallest rectangular *Grid* that contains all the data points that satisfy the given constraint. Further, the rank of the *Grid* is not reduced. A four-dimensional *Grid*, when sampled with a selection clause, will still return a four-dimensional *Grid*, even if some of the dimensions are of length one.

Sequence A *Sequence* is always returned as a *Sequence*, even if a selection clause selects only a single entry or no entry at all. If a projection clause identifies more than one member of the *Sequence*, they MUST be returned in the same *Sequence*.

NOTE:

What about allowing selection based on *Attribute* content? Say, all variables with origin="helena"? All variables with a *Attribute* named "units"? All variables with an *Attribute* named "units" whose value is "cm"?

This might be difficult, but well worth the effort.

5 Client/Server Interaction

The DAP is based on the request/response paradigm for client-server interaction.[10] This section provides an overview of the requests and responses (*i.e.*, the messages) which DAP-compliant servers MUST support. These messages are used to request information about the capabilities of a server, to request information about a particular data source made accessible by a server as well as requesting data values from the server. Messages which access a particular data source use the previously described data model. Other messages use simpler documents with contents which do not need a formal abstract definition.

The table below provides a description of the DAP messages. The precise details of the requests and responses are described in Section 6 (page 18) and Section 7 (page 41) . The mechanism used to communicate those requests and responses to/from a client and server depend on the transport protocol in use. (Consult ***DAP Web Services Specification*** for an HTTP implementation.) But whatever the protocol, a server MUST be able to provide the responses outlined in Table 7.

Table 7: DAP Requests and Responses

Requests	**Response**
Data, MAY include a constraint expression (Section 4 on page 12)	DDX (Section 6 on page 18)
Data (binary)	Blob (Section 6.5 on page 36)
Characteristics of server	Capabilities document (Section 4.1.3 on page 16)
	ErrorX object (Section 6.6 on page 39)

For a client to get data from a server takes at minimum two exchanges, first to request the DDX, and second to get the Blob. The DAP is at root a stateless protocol. The server is not required to remember anything from one request to another. A client has the responsibility to make the two requests correspond.

If a constraint expression is included in a request for a DDX, the returned DDX MUST contain a Blob reference that refers to the constrained data.

In addition to these data objects, a DAP server MAY provide additional "services" which clients may find useful. The HTTP implementation of the DAP, for example, provides HTML-formatted representations of a dataset's structure and a way to get data represented in CSV-style ASCII tables. These additional services are not described in this document; they are considered specific to different transport protocols and are described by the specifications for those particular protocols (such as the ***DAP Web Services Specification*** document).

5.1 Request and Response Information

The following information MAY be included in ANY request-response interaction between DAP a client-server pair. Because different transport protocols often provide ways to encode this type of information (*e.g.*, HTTP provides a way to encode the date of the response), a concrete syntax for representing this information is not presented here; that syntax MUST be included in the transport-specific DAP specification.

Compression DAP clients MUST be provided a way indicate to a server that they are able to process compressed responses. A server MAY compress a response ONLY if a client has indicated that it can process the compressed response. A server is NEVER under and obligation to compress a response.

Support for particular compression algorithms is specific to the transport protocol.

User agent DAP clients MAY provide information about the client software to the server. DAP servers MAY log this information. Note that DAP servers MUST provide their version and software information to clients and do so using a special response.

Date Servers MUST provide a date stamp which conforms to RFC 1033 in their responses, and they SHOULD also provide the last modification date, also conformant to RFC 1033, of the data requested.

[10] Should we add a reference to Fielding, "REST" here?

Byte Order DAP clients MUST be able to indicate their native byte order to a server. A server MAY choose to use little-endian byte order with a client that indicates that is its native byte order. By default the DAP uses big-endian byte order for all data exchanges. All DAP clients MUST be able to understand responses in big-endian byte order.

Floating-point Format Like the byte order, a client and server that agree on a floating-point format different than the IEEE 754 standard used by the DAP should be able to communicate that fact to each other, and skip converting data only to convert it back. A client MUST be able to indicate its preferred floating-point format; a server MUST be able to IEEE 754.

6 Responses

In order to pass data from server to requester, it needs to be transformed into a representation both can understand. In the same way that the idea behind a book needs to be written down and printed in order to transfer the idea from the writer to the reader, a dataset—an abstract set of numbers and the relationships between them—needs to be transformed into a more tangible form in order to be communicated.

For the DAP, this form is called the *persistent* representation.[11] This is to contrast it with the representations used within the memory of some program that can process this data, which of course only persist as long as the program is running. The persistent representation may also be contrasted with the file format used to store some data on a disk somewhere. File formats, though persistent, tend to be specific to particular machine architectures. The DAP needs a data representation that can be understood by all the clients and server programs likely to be used on it.

Under the DAP, there are four categories of information that pass from the server to the client: information about data, the data itself, error messages, and information about the server. The first three of these correspond to the three important DAP data objects: the DDX, Blob, and ErrorX objects. Information about a DAP server is provided by version messages and the Server Capabilities Document. These are described in detail in this section.

Some of the details about how DAP data objects are transmitted from server to client are specific to the communication protocol used. For these details relevant to the HTTP version of the DAP, see ***DAP Web Services Specification***. For a description of the data objects described by the DDX object, see Section 2 (page 4) and Section 3 (page 11) .

6.1 DDX

The DDX is an XML representation of the structure of all or part of a data set, as well as a description of the variables within that *Dataset*. A data set's structure is its constituent variables. Each variable, has a name, type, value, zero or more *Attributes* and an optional origin. The *Dataset* itself is modeled as a *Structure* variable and so has its own set of *Attributes* an origin. The values of variables are encoded in the Blob object, and are only indirectly part of the DDX object. *Attribute* values, however, are recorded directly in the DDX. The constraint expression mechanism can be used to request a DDX that does not contain any *Attribute* information (see Section 7 on page 41)

The DDX is intended to be a way for client programs to both learn about the contents of a data source and then to access some or all of the information held by that data source. It is possible to ask a DAP server to send a DDX which describes only a portion of the data source's complete content (*e.g.*, to send only one variable within the data source and to limit that to only the first 100 by 100 elements (see Section 4 (page 12) for information about constraint expressions). In normal operation a client program will ask a server for the DDX for an entire *Dataset*, determine the variables in which it is interested and then request a second DDX that contains only those variables using a constraint expression. It will ignore the Blob reference in the first DDX and use the Blob in the second to access the values it wants.

When clients access the information held in a data source's variable, they do so using the Blob. It is the Blob which is used to transfer that information from the data source, via a server, to the client. A DDX provides descriptive information about the data source, the names, types and *Attributes* held by the data source. The DDX also provides a reference to the Blob that holds that information. When a client requests the DDX for an entire data source, it is sent a DDX which contains a reference to the Blob which, in turn, will return the values of the all the variables. The client is under no obligation to access a Blob and the server need not 'create' it. The server's contract is simply that, if asked, it will return the Blob.

[11] Sometimes the persistent representation is called the *network representation* .

It is possible, using the constraint expression, to ask a data source to return a DDX which contains no *Attribute* information. Each DDX holds not only the name and type of each variable, but also its *Attributes*. *Attributes* do not have their values accessed using a separate document/object as variables do; their values are included in the DDX. If a client is to make several requests from a single data source, it is important to avoid the needless repetition of the *Attribute* information.

Example 1:

Here is an example DDX which contains a single two dimensional array. The *Dataset* also contains a single global *Attribute*:

```
<Dataset name="fnoc1.nc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.opendap.org/ns/OPeNDAP"
    xsi:schemaLocation="http://www.opendap.org/ns/OPeNDAP
                        http://dods.coas.oregonstate.edu:8080/opendap/opendap.xsd" >

    <Attribute name="Description" type="String">
        <value>Fleet Numerical Wind Data</value>
    </Attribute>

    <Array name="u">
        <Attribute name="long_name" type="String">
            <value>U_Wind_Vector</value>
        </Attribute>

        <Float32/>

        <dimension size="16" name="latitude">
        <dimension size="17" name="longitude">
        <dimension size="21" name="time">
    </Array>

    <Blob URL="http://dcz.opendap.org/dap/data/nc/fnoc1.nc?u"/>
</Dataset>
```

NOTE: Each of the XML elements used to declare a variable has a `name` attribute. The XML Schema (the rigorous definition) for the syntax of the XML document declares that the `name` attribute is optional. In practice this is not the case, with one exception. Consider that, without a name, there is no way for a client to ask for a variable. The only exception to this is the template variable for an *Array*. Each *Array* has a single child element which declares the type of the *Array*. Naming the child element is redundant (see Section 6.3.2 on page 20), and if named the name will be ignored.

6.2 XML Schema

The syntax and rules for the DDX document are encapsulated (to the extent possible) in an XML schema. The XML schema language is not adequate to completely define and enforce the rules for the DDX. The description of the DDX elements in Section 6.3 (page 20) constitutes the complete list of rules and syntax for the DDX.

6.2.1 XML Schema Validation

The XML schema is used to validate DDX documents as part of parsing them into memory resident software entities. The act of validating an instance of the DDX against the schema guarantees that the DDX will fulfill all of the syntax rules encapsulated in the schema, thus allowing subsequent software in the processing chain to take as “true” a large number of facts about the content and structure of the instance of the DDX document.

OPeNDAP servers MUST validate their DDX documents before sending them in response to a request. OPeNDAP clients MAY validate the returned DDX document, but this is seen as non-essential as the servers should be providing correct DDX instances.

6.3 DDX XML Elements

This section contains the detailed syntax descriptions of all of the component elements of a DDX.

6.3.1 Alias

This element creates a second name for some dataset variable. References to the *Alias* will produce the same results as reference to the *Alias* target identified in the *source* attribute. See Section 2.4 (page 10) .

The following *Alias* declaration creates a second name for a variable named `pepper`, part of a structure named `spice`. With this declaration in place, you can refer to the same value with three different names: `/spice/pepper`, `/spice/poivre` and `/goeswithsalt`.

```
<Dataset name="test">
    <Structure name="spice">
        <Float64 name="pepper"/>
        <Alias name="poivre" source="/spice/pepper"/>
    </Structure>
    <Alias name="goeswithsalt" source="/spice/pepper"/>
</Dataset>
```

Element attributes:

`name` [Required] The name of the *Alias*.

`source` [Required] The name of the variable to which this *Alias* refers (the *target* variable). The name given here MUST be a *fully qualified name* (See Section 2.3.2 on page 10) and MAY NOT refer to a member variable of a *Sequence*. Similarly , if the *Alias* is member of a *Sequence*, then it MAY NOT refer to a variable outside of it's parent *Sequence*. See Section 2.4 (page 10) formore details.

`role` [Conditional] The role that the *Alias* will play as a member of a *Grid*. The only acceptable values are `array` and `map`. This attribute MUST used if the *Alias* is a member variable of a *Grid*, either the *Array* or the *Map*. This attribute MAY NOT be used if the *Alias* is not a member element of a *Grid*. See Example 13 on page 29 and Example 14 on page 29 for usage examples.

`origin` [Conditional] The `modifier` that added this *Alias* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12)

Child elements: None.

6.3.2 Array

Declares an *Array* variable. See Section 2.2.1 (page 7) for a description of *Arrays*.

Element attributes:

`name` [Required] The name of the *Array*.

`origin` [Conditional] The `modifier` that added this *Array* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12)

Child elements:

dimension [Required] Each *dimension* element corresponds to a dimension of the *Array*. The order of the *dimension* elements indicates the order of the dimensions of the *Array*. As is the case with ANSI C[13] and C++[14], the rightmost/last dimension varies fastest. At least one *dimension* element is REQUIRED; there is no upper bound on the number of dimensions.

Attribute [Optional] The *Attributes* for this *Array*.

Template variable declaration [Required] An *Array* MUST have one *template variable* which MAY be any type with the exception that it MUST NOT be an *Alias*, an *Enumeration*, or an *Array* itself. The *name* attribute of the *template variable* is OPTIONAL, and in fact will be ignored if used. The name of the *Array* is defined by the *name* attribute of the *Array* element. The template variable is not directly accessible through the data model.

Child element syntax:

- Zero or more *Attributes* followed by
- One template variable element followed by
- One or more *dimension* elements.

Here is an array of structures:

Example 2:
```
<Array name="time_series">
  <Structure>
    <Float64 name="X_velocity"/>
    <Float64 name="Y_velocity"/>
  </Structure>
  <dimension size="72"/>
</Array>
```

6.3.3 Attribute

Use this element to attach *Attribute* values to a variable. Every variable MUST have a name, type, and value. Beyond that, it can have an arbitrary number of *Attributes*. See Section 3 (page 11) for more detail about what *Attributes* are and how they are used.

If a variable element has *Attribute* elements, then the *Attribute* elements MUST immediately follow the variable element opening tag.

An *Attribute* MAY contain multiple values of the same type (in essence a 1-dimensional array), or it MAY contain other *Attributes*. It MAY NOT contain both values and *Attributes*.

In other words: An *Attribute* MAY contain 1 or more homogeneously typed value elements, OR it MAY contain 1 or more *Attribute* elements. Example 3 on page 22 contains a syntax example of *Attributes*.

Attribute Aliases

Attribute elements MAY have a type of *alias*. An *alias* typed *Attribute* MAY refer to any other *Attribute* in the dataset. If an *Attribute* is of type *alias* it's XML declaration MUST have a *source* attribute whose value is the *fully qualified name* in the dataset of the *Attribute* to which the *alias* refers. See Section 2.3.2 (page 10) . There are specific rules for encoding the value of the `source` attribute. Rules for encoding the value of the source attribute MUST be applied prior to encoding the content for it's XML representation. See Section 6.4 (page 35) for the details of this encoding.

Example 4 on page 23 contains a syntax example for `alias` typed *Attributes*.

Processing Attributes

Attribute elements MAY have a type of *modifier*. *Attributes* of type *modifier* MUST be global (exist at the top level of the *Dataset*) and MAY not be used as members of other *Attributes* or of other variable *Attribute* structures. See Section 3.2 (page 12)

Attributes of type `modifier` MUST contain the following elements:

origin_server An *Attribute* of type *URL* whose name MUST be `origin_server` and whose value is the *URL* for the original data source.

organization An *Attribute* of type *String* whose name MUST be `organization` and whose value MUST be the name or the description of the organization that is providing the modified data.

modifying_service An *Attribute* of type *URL* whose name MUST be `modifying_service` and whose value MUST be the URL of the service that introduced the changes.

A syntax example of a `modifier` typed *Attribute* can be found in Example 5 on page 23.

Element attributes:

`name` [Required] A string containing the name of the *Attribute*.

`type` [Required] The type of this *Attribute*'s value. This MUST be one of the following:

- `Boolean`
- `Byte`
- `Int16`
- `UInt16`
- `Int32`
- `UInt32`
- `Int64`
- `UInt64`
- `Float32`
- `Float64`
- `String`
- `URL`
- `modifier`
- `alias`
- `structure`

An *Attribute* of type structure has a syntax comparable to that of a *Structure* variable. *Binary Images* and *Enumerations* are not permitted *Attribute* types.

`source` [Conditional] A string containing the name of the source *Attribute* for an *Attribute* of type `alias`. Used only of the *Attribute* is of type `alias`.

`origin` [Conditional] The `modifier` that added this *Attribute* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12)

Child elements:

value [Conditional] One or more child *value* elements, allowed ONLY if there are no child *Attribute* elements.

Attribute [Conditional] One or more child *Attribute* elements, allowed ONLY if there are no child *value* elements.

Child element syntax:

- One or more *Attribute* elements

OR

- One or more *value* elements

Example 3:

Here are examples of the *Attribute* element syntax.

```
<Dataset name="test" >
    <Structure name="measurement">
        <Attribute name="date" type="String">
            <value>18 Mar 03</value>
        </Attribute>
        <Attribute name="other" type="Structure">
            <Attribute name="satellite_name" type="String">
                <value>GOES</value>
            </Attribute>
            <Attribute name="experiment number" type="int32">
                <value>986743</value>
            </Attribute>
            <Attribute name="team" type="String">
                <value>Baker</value>
                <value>Charlie</value>
                <value>Dogg</value>
            </Attribute>
        </Attribute>
        <Float64 name="value">
        <Array name="time_series"
            <dimension size="32">
        </Array>
    </Structure>
</Dataset>
```

Example 4:

This example shows the use of `alias` typed *Attributes*.

```
<Dataset name="test" >
    <Structure name="measurement">
        <Attribute name="team" type="structure">
            <Attribute name="lead engineer" type="String">
                <value>Chet Baker</value>
            </Attribute>
            <Attribute name="software engineer" type="String">
                <value>Charlie Parker</value>
            </Attribute>
            <Attribute name="electrical engineer" type="String">
                <value>Ozzy Osbourne</value>
            </Attribute>
        </Attribute>
        <Float64 name="value">
        <Array name="time_series">
            <Attribute name="author"
                       type="alias"
                       source="/measurement/team/software engineer" />
            <dimension size="32">
            <FLoat32/>
        </Array>
    </Structure>
</Dataset>
```

Example 5:

In this example of the use of a `modifier` *Attribute*, the AIS server at ais.gso.uri.edu added a variable called `sst`, and an *Attribute* called ”units” to the variable called `Depth`.

```
<Dataset name="test">
    <Attribute name="helena" type="modifier">
        <Attribute name="origin_server" value="http://dods.gso.uri.edu/cgi/nph-nc"/>
        <Attribute name="modifying_service" value="http://ais.gso.uri.edu"/>
        <Attribute name="organization" value="URI/GSO"/>
    </Attribute>
      .
      .
      .
    <Float64 name="Depth">
        <Attribute name="units" type="String" origin="helena">
            <value>meters</value>
        </Attribute>
    </Float64>
      .
      .
      .
    <Array name="sst" origin="helena">
      .
      .
      .
    </Array>
</Datset>
```

6.3.4 Binary

Declares a *Binary Image*. This is an atomic type of arbitrary size with an undeclared internal structure. See Section 2.1.6 (page 6) .

The size of a binary image is encoded in the Blob object, so declaring it in the *Binary* element is optional. For a constructor which holds *Binary Images*, declaring the size here is a convenience for clients and is OPTIONAL. Doing so can increase the efficiency of clients who have to deal with the data after downloading it. If the size of a *Binary Image* which appears in a Constructor type such as *Sequence* is declared, all instances MUST be that size.

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`size` [Optional] The size of the image, in bytes.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Binary Image*.

Child element syntax:

• Zero or more *Attribute* elements

Example 6: `<Binary name="sound_sample" size="17256"/>`

6.3.5 Blob

This is the reference to the serialized binary data content described by this DDX. See Section 6.5 (page 36) .

Element attributes:

`URL` [Required] A string containing the web address of the Blob object associated with this DDX.

Child elements: None

See Example 1 on page 19.

6.3.6 Boolean

A variable that takes on one of two values: `true` or `false`. See Section 2.1.2 (page 5) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Boolean*.

Child element syntax:

• Zero or more *Attribute* elements

Example 7:
```
<Boolean name="QC">
  <Attribute name="long_name" type="String">
    <value>Quality Control Flag</value>
  </Attribute>
</Boolean>
```

6.3.7 Byte

Declaration of an eight-bit unsigned integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Byte*.

Child element syntax:

• Zero or more *Attribute* elements

Example 8:
```
<Byte name="Temperature">
  <Attribtue name="units" type="String">
    <value>Counts</value>
  </Attribute>
</Byte>
```

6.3.8 Dataset

A *Dataset* element contains all the variable and global attribute for a data source. The DDX always contains a *Dataset* element as its root. A *Dataset* element is semantically equivalent to the *Structure* variable; the rules for encoding the variables in a *Structure* apply to the variables at the top level of the *Dataset* element. See Section 2.2.2 (page 8) for information about the semantics of a *Structure*. See Section 3 (page 11) about global and other *Attributes*.

Element attributes:

`name` [Required] A string containing the name of the variable.

`xmlns` [Required] A URI containing the default namespace declaration for the XML content of the *Dataset* document. At the time of this writing this value should be *http://www.opendap.org/ns/OPeNDAP*

`xmlns:xsi` [Required] Maps the namespace identifier `xsi` to the URI provided in the value. In this case it should always be set to *http://www.w3.org/2001/XMLSchema-instance*

`xsi:schemaLocation` [Required] Should be set to a pair values containing the default namespace URI followed by a URL that when dereferenced will provide the schema for the namespace. The schema location URL will typically be set to a location on the server that is providing the *Dataset* document.

`origin` [Conditional] The "modifier" that created this *Dataset*. This would be used to indicate that the *Dataset* was created by an Aggregation Server of some sort. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Dataset*.

Variable Element Declarations [Required] The collection of variables present in the dataset

Child element syntax:

- Zero or more *Attribute* elements; followed by
- One or more variable elements; followed by
- One Blob element

See Example 1 on page 19.

6.3.9 dimension

This element appears within *Array* and *Map* declarations, and declares the length (and possibly the name) of a dimension. For multidimensional *Arrays* or *Maps*, the first *dimension* element corresponds to the left-most *Array* or *Map* index, *et cetera*.

Element attributes:

`name` [Optional] A string containing the name of the dimension.

`size` [Required] The number of elements in the dimension under consideration.

Child elements: None.

See Example 2 on page 21.

6.3.10 Enumeration

An *Enumeration* is used to bind symbols to a set of discrete integral values. Each element of an *Enumeration* is called an *enumerator*.

Element attributes:

`name` [Required] A string containing the name of the variable.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*.

Child elements:

enum [Required] Used to hold a discrete value of the *Enumeration*. Each *Enumeration* MUST have at least one *enum* element.

Attribute [Optional] The *Attributes* for this *Enumeration*.

Child element syntax:

- Zero or more *Attribute* elements; followed by
- One or more *enum* elements

Example 9:
```
<Enumeration name="error_codes">
    <enum name="no_such_file" value="0"/>
    <enum name="insufficient_permissions" value="1"/>
    </Enumeration>
```

6.3.11 enumerator

This element holds a single enumerator for an *Enumeration* element.

Element attributes:

`name` [Required] The name of this *enumerator*.

`value` [Required] The integral value of this *enumerator*. Limited to $4,294,967,296$ values (2^{32}).

Child elements: None.

See Example 9.

6.3.12 Float32

Declares an IEEE 754 conformant data variable to hold a 32-bit floating-point value. See Section 2.1.4 (page 6) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Float32*.

Child element syntax:

- Zero or more *Attribute* elements

Example 10: `<Float32 name="Temperature"/>`

6.3.13 Float64

Declares an IEEE 754 conformant data variable to hold a 64-bit floating-point value. See Section 2.1.4 (page 6) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Float64*.

Child element syntax:

• Zero or more *Attribute* elements

Example 11:
```
<Float64 name="Temperature">
    <Attribtue name="units" type="String">
      <value>Degrees_Kelvin</value>
    </Attribute>
</Float64>
```

6.3.14 Grid

Declares a *Grid* variable (see Section 2.2.3 on page 8). This is comparable to an *Array* (in fact it contains an *Array*), but there is somewhat more freedom in assigning coordinates to any point in the *Array*. Unlike an *Array*, a *Grid* can be indexed using types other than integers. The mapping between different values and discrete elements of the *Grid* is given by the *Maps*. Within a *Grid* the correspondence between any dimension of the *Grid*'s target *Array* and a *Map* is made by insuring that corresponding target *Array* dimensions and *Map* dimensions have the same name and size. A *Map* is a type of an *Array* and so has its own dimension element, which MUST be named, and this name is used to create the binding of the *Map* to a dimension of the *Grid*'s target *Array* with the same name. While the naming of the dimensions establishes the relationship between *Maps* and *Array* components in a *Grid*, the dimensions MUST be the same size in order for the mapping to be complete. If two dimensions in a *Grid* of the same name do not have the same size an error will be generated.

In order to accommodate the re-use of the components of a *Dataset* and streamline the transmission of data by through redundancy reduction both the target *Array* and the *Map* elements of a *Grid* MAY be replaced with an *Alias* variable. In both cases the *Alias* MUST refer to an *Array* type (either an *Array* or a *Map*) somewhere else in the *Dataset*. The rules about naming and size for the constituent dimensions of the *Grid* components will be evaluated against the source of the *Alias* reference as if they where actually components of the *Grid*. (see the comments in Example 12 on page 29, Example 13 on page 29, and Example 14 on page 29).

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Grid*.

Map [Required] The maps for this *Grid*. These can be thought of as the independent variables for the *Grid*'s *Array*. There MUST be at least one map and no more than N where N is the rank of the *Grid*'s *Array*.

Child element syntax:

- Zero or more *Attribute* elements; followed by
- Zero or one *Array* element; followed by
- Zero or more *Map* elements; followed by
- Zero or more *Alias* elements.

Each dimension of the target *Array* MUST have one or more corresponding *Maps* dimensions of the same name and size. If a *Map* has multiple dimensions, then each one MUST be a valid *Map* dimension for one of the dimensions in the target *Array*. An *Alias* that refers to an *Array* of the correct size and shape MAY replace any or all of *Map* elements and/or the *Array* element.

Here's a *Grid* containing a two-dimensional target array and two one-dimensional map arrays.

Example 12:

This example shows a typically *Grid*. All of the *Grid* components, the target *Array* and the *Maps*, are declared as members of the *Grid* element.

```
<Grid name="v">
    <Array name="temp">
        <Byte/>
        <dimension name="lat" size="5"/> <!-- bound to the Map dimension named "lat" -->
        <dimension name="lon" size="5"/>
    </Array>
    <Map name="y">                  <!-- this name does not matter -->
        <Float64/>
        <dimension size="5" name="lat"/>  <!-- this name completes the association -->
    <Map name="x">
        <Float64/>
        <dimension size="5" name="lon"/>
    </Map>
</Grid>
```

Example 13:

This example shows a *Grid* where the target *Array* element is actually an *Alias* to an *Array* outside of the *Grid* element.

```
<Array name="temp">
    <Byte/>
    <dimension name="lat" size="5"/> <!-- bound to the Map dimension named "lat" -->
    <dimension name="lon" size="5"/>
</Array>
<Grid name="v">
    <Alias name="sst" source="/temp" role="Array" />
    <Map name="y">                  <!-- this name does not matter -->
        <Float64/>
        <dimension size="5" name="lat"/>  <!-- this name completes the association -->
    </Map>
    <Map name="x">
        <Float64/>
        <dimension size="5" name="lon"/>
    </Map>
</Grid>
```

Example 14:

This example shows a *Grid* where all of the member elements (the target *Array* and the *Maps*) are actually an *Aliases* to a *Arrays* outside of the *Grid* element.

```
<Array name="temp">
    <Byte/>
    <dimension name="lat" size="5"/> <!-- bound to the Map dimension named "lat" -->
    <dimension name="lon" size="5"/>
</Array>
< Array name="y">                   <!-- this name does not matter -->
    <Float64/>
    <dimension size="5" name="lat"/>  <!-- this name completes the association -->
</Array >
< Array name="x">
    <Float64/>
    <dimension size="5" name="lon"/>
</Array >
<Grid name="v">
    <Alias name="sst" source="/temp" role="Array" />
    <Alias name="NS" source="/y" role="Map" />
    <Alias name="EW" source="/x" role="Map" />
</Grid>
```

6.3.15 Int16

A 16-bit signed (twos-complement) integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Int16*.

Child element syntax:

• Zero or more *Attribute* elements

Example 15: `<Int16 name="Temperature"/>`

6.3.16 Int32

A 32-bit signed (twos-complement) integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Int32*.

Child element syntax:

• Zero or more *Attribute* elements

Example 16: `<Int32 name="Temperature"/>`

6.3.17 Int64

A 64-bit signed (twos-complement) integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Int64*.

Child element syntax:

• Zero or more *Attribute* elements

Example 17: `<Int64 name="Temperature"/>`

6.3.18 Map

This declaration creates a *Map* used in a *Grid*. See Section 2.2.3 (page 8) for a description of *Grids*. Also see Example 12 on page 29, Example 13 on page 29, and Example 14 on page 29. A *Map* is a special case of the Array element; it has the same *Attributes* and child elements but MAY only appear inside a *Grid*.

Element attributes:

`name` [Required] A string containing the name of the *Map* variable. The relationship between the dimension(s) of the *Map* and the target *Array* of the *Grid* is established through the names of the *Map* dimensions. The name of the *Map* is not used to establish the relationship, but should be used in some informative manner for the user. See Example 12 on page 29.

Child elements:

dimension [Required] Each *dimension* element corresponds to a dimension of the *Map*. The order of the *dimension* elements indicates the order of the dimensions of the *Map*. As is the case with ANSI-C[13] and C++[14], the rightmost/last dimension varies fastest. At least one *dimension* element is REQUIRED; there is no upper bound on the number of *dimension* elements

Attribute [Optional] The *Attributes* for this *Map*.

Template variable declaration [Required] A *Map* MUST have one *template variable* which MAY be any type except that it MUST NOT be an *Alias* or an *Array* itself. The *name* attribute of the *template variable* is optional, and in fact will be ignored if used. The name of the *Map* is defined by the *name* attribute of the *Map* element. The template variable is not directly accessible through the data model.

Child element syntax:

• Zero or more *Attributes* followed by
• One template *variable* element followed by
• One or more *dimension* elements.

6.3.19 Sequence

A *Sequence* is an ordered set of entries, each instance of which is comparable to a *Structure* variable. Each entry in a *Sequence* contains the same set of variables. A *Sequence* can also be thought of as a relational database table, with each entry corresponding to a single row.

See Section 2.2.4 (page 9) for a description of the *Sequence* type.

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Sequence*.

Variable declaration [Required] A *Sequence* MUST have one or more *Variable declarations.* They MAY be any type of variable.

Child element syntax:

- Zero or more *Attributes*; followed by
- One or more *variable* elements

Example 18:
```
<Sequence name="gallimaufry">
  <Float64 name="measurement"/>
  <Array name="measurement_collection">
    <Int16/>
    <dimension size="32">
    <dimension size="45">
  </Array>
</Sequence>
```

6.3.20 String

A series of Unicode (UTF-8) characters. See Section 2.1.5 (page 6) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *String*.

Child element syntax:

- Zero or more *Attributes*

Example 19: `<String name="Name"/>`

6.3.21 Structure

An ordered set of variables. See Section 2.2.2 (page 8) for a description.

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Structure*.

Variable declaration [Required] A *Structure* MUST have one or more *Variable declarations*. They MAY be any type of variable.

Child element syntax:

- Zero or more *Attributes*; followed by
- One or more *variable declarations*

Example 20:
```
<Structure name="person">
  <String name="name">
  <Float64 name="height">
  <Int32 name="age">
</Structure>
```

6.3.22 UInt16

An unsigned 16-bit integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *UInt16*.

Child element syntax:

- Zero or more *Attributes*

Example 21: `<UInt16 name="Temperature"/>`

6.3.23 UInt32

An unsigned 32-bit integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *UInt32*.

Child element syntax:

• Zero or more *Attributes*

Example 22: `<UInt32 name="Temperature"/>`

6.3.24 UInt64

An unsigned 64-bit integer. See Section 2.1.1 (page 4) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *UInt64*.

Child element syntax:

• Zero or more *Attributes*

Example 23: `<UInt64 name="Temperature"/>`

6.3.25 Url

A variable which contains a URL. A *URL* MUST only contain characters that are legal parts of an internet URL. Currently, this is limited to single-byte US-ASCII (7-bit) characters. See Section 2.1.5 (page 6) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Url*.

Child element syntax:

• Zero or more *Attributes*

Example 24: `<URL name="SST_data_server"/>`

6.3.26 Time

A variable which contains an ISO 8601 time string[12]. A *Time* can contain only characters that legal for an ISO 8601 time string. Currently, this is limited to single-byte US-ASCII (7-bit) characters. See Section 2.1.5 (page 6) .

Element attributes:

`name` [Conditional] A string containing the name of the variable. May (and should) be omitted if the element is being used as the *template variable declaration* for an *Array*.

`origin` [Conditional] The `modifier` that added this *Binary Image* to the document. Used only if this element was added in conjunction with a *Processing Attribute*. See Section 3.2 (page 12) .

Child elements:

Attribute [Optional] The *Attributes* for this *Time*.

Child element syntax:

• Zero or more *Attributes*

Example 25: `<Time name="Sample_time"/>`

6.3.27 value

Use this tag to identify the value of an *Attribute* element. See Example 3 on page 22.

Variable values are recorded in the Blob object, while *Attribute* values are recorded directly in the DDX. This means that *Attribute* values MUST be able to be represented in a text file.

Element attributes: None.

Child elements: None.

Example 26:
```
<value>3.1415</value>
<value>2.718</value>
```

6.4 Encoding Rules

[*This section is a first stab, we expect that as we implement code to support this specification we will have to change (and add to) the Encoding Rules.*]

There are several encoding schemes that need to be applied to various XML components of the DAP. At the end of the list is the encoding for XML. Other encodings MUST be applied to different parts of the DDX.

6.4.1 *Attribute* and variable *Alias* source attribute encoding

The `source` attribute a variable *Alias* and the `source` attribute of an *Attribute* of type `alias` are both *fully qualified names*. Since *fully qualified names* use the slash ("/") character as the delimiter between variable (and *Attribute*) names it is necessary to specially identify this character when it appears as part of a legitimate variable (or *Attribute*) name in the DDX document (which must eventually be parsed by software). This identification is called "escaping the character", and is achieved by using a different character that will be interpreted to have the meaning of "escape". The backslash ("\") is the escape character. The the slash ("/") and backslash ("\") MUST be escaped if they appear as part of a node name in the absolute path. This encoding must applied prior to any additional encoding needed to make the representation XML compatible, and must persist after XML decoding.

6.4.2 *Project* element variable encoding

This is probably the same as Section 6.4.1 (page 35) .

6.4.3 *Select* element target encoding

This is probably the same as Section 6.4.1 (page 35) .

6.4.4 Base 64 *Attribute* value encoding

In order to decrease the total number of bytes in a DDX response it is possible that we may choose to transmit *Attribute* values in their serialized binary form (see Section 6.5 on page 36). To do so, and still include them in the DDX, we will have to encode them into a form that allows them to be represented as UTF-8. Most likely this will achieved by using a base-64 encoding.

6.4.5 XML document encoding

After all of the previous encodings have been applied to the appropriate parts for the document the document must be brought into a correct encoding for XML. This means that any content (as opposed to XML syntax characters) must be so encoded. This includes the values of XML element attributes and the content of XML elements. This encoding is minimally:

- The < (less than) character is replaced with <
- The > (greater than) character is replaced with >
- The & (ampersand) character is replaced with &
- The ’ (apostrophe) character is replaced with '
- The ” (double quote) character is replaced with "

And may in fact be more extensive. See [**?**] for more detailed information.

6.5 Blob

The Blob contains the serialized data represented by the DDX. XML documents like the DDX cannot efficiently transmit binary data [**?**], so a DAP DDX simply references an object that contains the data described in the DDX. The Blob is a way of accessing the data ‘out-of-band’ with respect to the DDX. In addition to eschewing the problems of processing large volumes of binary data with an XML parser, out-of-band access enables servers to support streaming in conjunction with protocols such as SOAP[**?**].

Data values associated with variables appear in the Blob in the order in which the variables are declared in the associated DDX. The DDX always contains a *Dataset* element as its root. A *Dataset* element is semantically equivalent to the *Structure* variable; the rules for encoding the variables in a *Structure* apply to the variables at the top level of the *Dataset* element.

The Blob is *not* self-documenting. A client program will be unable to make sense of it without the declarations in the accompanying DDX, since the variable types, sizes, and ordering determined from the structure/organization of the associated DDX.

NOTE: The DDX may be used to define a C++, Java, *et c.*, object which may then be used by a DAP client to allocate memory for the variables it declares. A DDX which has been created, but before the Blob object has been received is said to be an ”empty” DDX. After the Blob arrives and its data has been decoded and parceled out to the memory in the DDX, the DDX is said to be ”full.”

6.5.1 Length Specification: Representing lengths of encoded data elements

When values are encoded for transmission, they are often preceded by length information. For example, an *Array* is prefixed by the number of elements in the *Array*. Instead of representing this information in a fixed-size integer data type, the DAP encodes length information in a way that allows any length to be represented. These are called DAP *Length Specifications* and are defined as follows:

> Each byte of the length specification is divided into two parts: the high-order bit indicates whether another byte follows, and the low-order seven bits provide the length. A two-byte specification of a 500-byte image would be 10000011 01110100 (0x83 0x74), for example, while a 20-byte image would only need one byte of length information: 00010100 (0x14). Length specifications can use any number of bytes.

Length Specifications are used to specify the size of variable length entities throughout the DAP's XML encoding.

6.5.2 Blob framework and reliable error delivery

The Blob response is a multi-part MIME document[**?**]. The Blob MUST contain the `Content-Type` MIME header and MUST give the content type as `multipart/mixed`. Each part of the multi-part document MUST have a `Content-Length` header which indicates the total number of bytes in this part and MUST have a `Content-Type` MIME header and the type of that header MUST be `application/octet-stream` or it MUST `text/plain`. The `application/octet-stream` indicates that binary data are contained in this part of the document and are to be interpreted as described in Section 6.5.3 (page 37) and Section 6.5.4 (page 37) . If the type is `text/plain`, then the content is assumed to be an ErrorX document.

The Blob response is encoded using a multi-part MIME document to ensure reliable delivery of error messages if servers stream responses to clients. A server can iteratively build chunks of the total response and include that in the Blob as the next part. If an error is discovered, that can be sent instead. A server is free to choose the size for each part and is free to build Blob responses with only one part.

6.5.3 Atomic Types

The DAP atomic data types are encoded as follows. All data are encoded using big-endian byte order unless that is modified using the client-server negotiation options described in Section 5.1 (page 17) .

Integer types Signed integers: Twos-complement; unsigned integers: straight binary. The *Boolean* and *Enumeration* types are sent as unsigned 32-bit integers.[12]

Floating-Point types IEEE 754

String types *Strings*: Unicode, UHF-8, prefixed by a *Length Specification* indicating the length *in bytes* of the string. The *URL* and *Time* types are limited to US-ASCII characters, prefixed by a *Length Specification* indicating the length *in bytes*.

Binary images A *Binary Image* is sent as a sequence of bytes, prefixed by a *Length Specification* indicating the length *in bytes* of the *Binary Image*.

6.5.4 Constructor Types

The constructor types are encoded as follows. See above for instructions about how to encode the atomic types.

Array *Array* members are encoded in row-major order (rightmost subscript varies fastest). The *Array* is preceded *Length Specification* indicating the the number of *elements* (not the number of bytes) in the *Array*. The size of each element MUST be derived from the declaration in the DDX. *Arrays* of *String*, *URL* or *Time* values MUST include the *Length Specification*.

[12] The *Boolean* type could conceivably be sent in a single bit, but decoding that may be inefficient for some architectures; if a transport protocol supports compressions that may achieve the same reduction in size.

Structure Members are placed in order of declaration, with no boundary values. *Alias* variables are omitted. (There is no place holder for them in the Blob. They are reconstructed using the DDX.)

Grid *Grids*, which are essentially special cases of the *Structure* type, are recorded in the same fashion as *Structures*.

Sequence Each entry in a *Sequence* is recorded like a *Structure*, except that each entry is preceded by a single "Start-of-Instance" flag byte (value `0x5a`), and the entire *Sequence* is ended with an "End-of-Sequence" flag byte (same value: `0x5`).

For example, a schematic view of the data and flags for the three element sequence:

```
<Sequence>
  <Int32 name="Var1">
  <Int32 name="Var2">
  <Int32 name="Var3">
</Sequence>
```

looks like:

```
<SOI><Var1><Var2><Var3>
<SOI><Var1><Var2><Var3>
<SOI><Var1><Var2><Var3>
<EOS>
```

For a nested Sequence (a Sequence which contains a Sequence as a child element) such as:

```
<Sequence>
  <Int32 name="Var1">
  <Int32 name="Var2">
  <Sequence>
    <Int32 name="Var3">
    <Int32 name="Var4">
  </Sequence>
</Sequence>
```

the schematic representation looks like:

```
<SOI><Var1><Var2>


<SOI><Var3><Var4>
<SOI><Var3><Var4>
<EOS>


<SOI><Var1><Var2>


<SOI><Var3><Var4>
<SOI><Var3><Var4>
<EOS>


<SOI><Var1><Var2>


<SOI><Var3><Var4>
<SOI><Var3><Var4>
<EOS>
<EOS>
```

Note that the outer sequence has three instances and each of those includes the inner Sequence.

6.6 ErrorX

The ErrorX object is an XML document containing information about any errors that may have been encountered by the server while processing a request. For any request, a server MAY return an ErrorX response in place of the normal response (*e.g.*, instead of the DDX).

The ErrorX object MUST contain:

Offending request information This is the complete URL, including payload (constraint expression), or the POST data in effect at the time. The intent is that there should be enough information to reproduce the error.

Text message A description of the problem.

The ErrorX object is an XML document used to signal a DAP client that the server has encountered an error of some kind.

6.7 ErrorX XML Elements

An ErrorX object can contain the following XML elements.

6.7.1 Error

Describes the type of error encountered. The element MUST contain a short text description with the *description* attribute, OR a longer description enclosed.

Element attributes:

`code` A number, from a set of well know error numbers, associated with this error. See Appendix Section B (page 46) for a list error numbers and their meanings. [required]

Child elements:

request [Required] Contains the Base URL given that triggered the error.

description [Optional] Contains a short description of the error condition.

constraint [Optional] Contains the constraint condition of the request that triggered the error. This is a string containing an index value corresponding to the index value of the *Constraint* element.

Child element syntax:

- One *request* element; followed by
- Zero or One *description* element; followed by
- Zero or One *constraint* element

Example 27:
```
<Error code="404">
  <description>Not found</description>
  <request>http://dods.org/data.nc</request>
  <constraint>\emph{huh? What is the index element stuff?}</constraint>
</Error>
```

6.8 Server Capabilities Document

NOTE: We're going to be careful about the name 'Capabilities' since OpenGIS may have trademarked that. 10/21/03 jhrg

A DAP server MUST be equipped to respond to a client request for an XML document describing the characteristics and capabilities of that server.

The Server Capabilities Document MUST contain information about the DAP version. It MAY contain software implementation version information. The Server Capabilities Document MUST contain a description of ANY constraint expression function which is intended to be publically accessible (servers are free to include constraint expression functions for internal/experimental use and not document them).

6.9 Server Capabilities Document XML Elements

The XML syntax of the returned capabilities document is as follows:

6.9.1 Description

Use this element to include documentation too long to include as an attribute to the *Function* element.

Element attributes: None.

Child elements: None.

6.9.2 Function

Use the *Function* declaration to identify a function a client program can use in a constraint expression. The server identifies any part of the constraint expression that looks like `function(arg1,arg2,arg3)` as a function. If a function has no arguments, an empty set of parentheses MUST be included.

If the *Function* element has arguments, their declarations will be contained in its declaration. There should also be a text description of the Function. Short descriptions MAY be included as a Function attribute, while longer ones MAY be included in a *Description* element in the *Function* body.

Element attributes:

`name` [Required] A string identifying the name of the function. You should try to pick a name unlikely to cause confusion. The best idea is to pick a brief acronym with which to identify your server or project, and prefix all function names with those letters. That is, don't call your new function `exp()`. Instead call it something like `GDSexp()`.

`type` [Required] The type of data returned by the function. This is one of the DAP data types (Section 2 on page 4). Functions to be used in the selection clause of a constraint expression should return *Boolean*.

Child elements:

Description [Optional] Used to provide a detailed description of the function.

Parameter [Optional] Used to describe an input parameter for the function.

Child element syntax:

- Zero or one *Description*; followed by
- Zero or more *Parameters*

6.9.3 Parameter

Use the *Parameter* element to list the arguments (in order) needed by this function.

Element attributes:

`name` [Optional] A name by which to refer to (formal) parameter. This is a documentation convenience, and there is no default value.

`type` [Required] The type of the parameter. This is one of the DAP data type names (Section 2 on page 4). Note that the value of the `type` attribute is just the name of the type (*e.g.*, `Grid`, `Array`, *et c.*) and does not include information about the size of an Array or the names of the fileds in a Structure. Because it simplies the interface to a function, it is possible to use the special words `SimpleType` and `AnyType` in place of one of the DAP type names. The word `SimpleType` means that the actual parameter MAY be of any of the atomic types. The word `AnyType` means that the actual parameter MAY be of any of the DAP data types.

`repeats` [Optional] If present and assign the value `true`, this attribute indicates that the parameter MAY appear once or any number of times greater than once. If not present the parameter MUST appear once.

`description` [Optional] A brief text description of this argument. This is a documentation convenience, and there is no default value.

Child elements: None.

Example 28:
```
<Function name="average" type="Float64">
  <description>This function averages a group of Float64
  values.</descritpion>
  <parameter type="Float64" repeats="true"/>
</Function>
```

Example 29:
```
<Function name="mean" type="Float64">
  <description>This function averages a group of values;
  it is more liberal about its arguments than average. It will return an
  ErrorX if called with non-numeric actual parameters.</descritpion>
  <parameter type="SimpleType" repeats="true"/>
</Function>
```

6.9.4 Version

The *Version* element indicates the version of some entity associated with the server in question. The Server Capabilities Document response MUST contain the version of the DAP; other version elements MAY be used to include the versions of other things (such as implementation software). Note that the versions of specific data sources SHOULD be included in the *Attributes* for those data sources, not in the version inforamtion returned here.

Element attributes:

`value` [Required] A string containing the version number or name.

`entity` [Required] A string containing the name of the specification/standard, software or data source to which this *Version* element refers. The string `DAP` is reserved to refer to the DAP and indicates the version of the protocol, not the implementation. Any other value is at the discretion of the software implementor (for software version) or server administrator.

Child elements: None.

Example 30:
```
<Version entity="DAP" value="4.0"/>
<Version entity="OPeNDAP_netCDF_server" value="6.4"/>
```

7 Constraint

The DAP uses a single *request document* to supply information to a server about a request for data. Recall in Section 4 (page 12) that the DAP uses constraint expressions to limit data accessed to specific variables, or parts of variables, in a *Dataset*.

Each constraint is broken into two clauses, the *projection clause* and the *selection clause* . Each of the clauses is further broken down into sub-clauses. A projection clause is simply a collection of one or more *Project* elements, and

a selection clause consists of one or more *Select* elements. If no *Project* elements are present (the projection clause is omitted), the server will assume that all the variables in the dataset are to be returned. If the selection clause is omitted, all instances (values) of the variables specified in the projection are returned.

It is reasonable to for the *Constraint* element to be empty, as this will cause the server to return the complete description (DDX) for the *Dataset*. From this the user would typically form the *projection clauses* and *selection clauses* to constrain the information. A second request would then follow, this time with a more complex *Constraint*.

A client can ask the server to omit the *Attribute* information from a DDX by adding a *NoAttributes* element to the *Constraint*. See Section 7.2 (page 44) for examples.

7.1 Constraint XML Elements

Following is a description of each element used in a constraint expression.

7.1.1 Constraint

This element contains the two component clauses of the constraint expression: the projection and the selection. The projection clause specifies which variables are to be returned, and the selection clause helps select among the variable values. See page 12.

Element attributes:

`name` [Required] An identifying string for the constraint expression.

Child elements:

NoAttributes [Optional] Informs server to strip all *Attribute* information from the returned DDX.

Project [Optional] Identifies a variable to be returned the client. See Section 7.1.4 (page 43)

Select [Optional] Specifies what conditions need to be met for an instance of the set of projected variables to be returned to the client. See Section 7.1.5 (page 44)

Child element syntax:

- Zero or one *NoAttributes* element; followed by
- Zero or more *Project* elements; followed by
- Zero or more *Select* elements

See Section 7.2 (page 44) for examples of complete constraint expressions.

7.1.2 Hyperslab

Use this element to measure off a rectangular subsection (sometimes called a *hyperslab*) of a *Grid*, *Array*, or *Sequence* variable. See Section 4 (page 12) . There MUST be only one *Hyperslab* element for each dimension of the *Grid*. If a *Hyperslab* element is missing, that dimension will be returned whole. See Example 34 on page 44.

A single *Hyperslab* element MAY also be used to subsample a *Sequence*.

Element attributes:

`dimension` [Optional] The name of the dimension to sample with the parameters given in this *Hyperslab* element. This only applies to *Grid* variables or *Arrays* with named dimensions. If the name is omitted, the order of the *Hyperslab* elements will be assumed to be the same as the order of the variable dimension declarations.

`start` [Optional] The first index to return. If omitted, zero is assumed.

`stop` [Optional] The last index to return. If omitted, the dimension maximum is assumed.

`stride` [Optional] Use the *stride* to skip *Grid* rows (or columns or hyperslabs). A *stride* of two returns every other row, three returns every third, and so on. If omitted, a value of one is assumed.

Child elements: None.

Example 31:
```
<Constraint>
    <Project variable="/temp">
        <Hyperslab dimension="time" start="1" stop="100" stride="5"/>
        <Hyperslab dimension="depth" start="20" stop="40"/>
    </Project>
</Constraint>
```

7.1.3 NoAttributes

This element is used to eliminate the *Attribute* content from the returned DDX. This is intended to be used by clients making multiple requests of the same *Dataset* and thus do not need to have the *Attribute* elements sent multiple times.

Element attributes: None. **Child elements:** None.

Example 32:

This constraint expression will return a DDX of the entire *Dataset* minus any *Attribute* information.

```
<Constraint>
    < NoAttributes />
</Constraint>
```

7.1.4 Project

This element identifies a variable to be returned to the client. This MAY be either a variable in the *Dataset*, or it MAY be calculated by a server function, whose value is returned as if it were data. Projection functions MUST be identified in the server's Server Capabilities Document.

Contains: Zero or more *Hyperslab* elements.

A *Project* element MUST have one of either a *variable* or a *function* but not both.

Element attributes:

`function` [Optional] A function invocation. The function's return value (which can take any DAP data type) is returned to the client. **Note:** When variables are passed to a function it MUST be done so using their *fully qualified name*. See Section 2.3.2 (page 10) for more on *fully qualified names*.

`variable` [Required] The *fully qualified name* of the variable to be returned. See Section 2.3.2 (page 10) for more on *fully qualified names*.

Child elements:

Hyperslab [Optional] Describes a sub-sample of a *Grid*, *Array*, or *Sequence* variable. See Section 7.1.2 (page 42)

Child element syntax:

• Zero or more *Hyperslab* elements

7.1.5 Select

Use this element to define the condition under which a *Sequence* instance or a *Grid* element is to be returned. A *Select* element specifies a relational operation and two operands to compare with it. See Example 33 and Example 35 to see how to select from a *Sequence* and a *Grid*, respectively. Note that the *Select* MUST explicitly state to which projected variable it applies (using the `target` attribute).

Contains: Zero or more *Hyperslab* elements.

Element attributes:

`condition` [Required] A relational expression or a function call. In the case of a function call, the function MUST return a *Boolean* type. In the case of a relational expression, the syntax MUST be: $operand_1 operator operand_2$ where $operator$ is one of: $=, !=, <, <=, >, >=, =\sim$ and are defined in Table 6. The operands $operand_1$ and $operand_2$ maybe variables of any of the atomic types, including fields, constants, or function calls which return atomic types. **Note:** When variables are passed to a function it MUST be done so using their *fully qualified name*. See Section 2.3.2 (page 10) for more on *fully qualified names*.

`target` [Required] The *fully qualified name* of the variable with which this selection criterion is to be evaluated. See Section 2.3.2 (page 10) for more on *fully qualified names*.

Child elements:

Hyperslab [Optional] Describes a sub-sample of a *Grid*, *Array*, or *Sequence* variable. See Section 7.1.2 (page 42)

Child element syntax:

• Zero or more *Hyperslab* elements

7.2 Constraint examples

Example 33:

This constraint expression is a simple request for temperature and salinity from a *Sequence Dataset*. This will return a *Sequence* containing temperature and salinity pairs where all the salinity values are above 34.0.

```
<Constraint>
    <Project variable="/sample/temp"/>
    <Project variable="/sample/salt"/>
    <Select condition="/sample/salt>34.0" target="sample"/>
</Constraint>
```

Example 34:

For sub-sampling gridded data, use the *Project* element to elaborate a projection clause. This constraint expression subsamples a *Grid*, and returns a smaller *Grid*, where the `lat` dimension has rows 1,3,5,7 and 9 of the original *Grid*, and the `lon` dimension has all the columns from 20 to 40 from the original.

```
<Constraint>
    <Project variable="/sst">
        <Hyperslab dimension="lat" start="1" stop="10" stride="2"/>
        <Hyperslab dimension="lon" start="20" stop="40"/>
    </Project>
</Constraint>
```

Example 35:

It is also possible to select from a *Grid*, based on the values of the map arrays. This constraint expression shows the selection of a *Grid* called `sst`. Assuming `sst` is a two-dimensional array with two one-dimensional maps, this constraint will return a *Grid* where all the `lat` values are above 24.0 and all the `lon` values are below -50.0. In addition this constraint expression requests that no *Attribute* information be sent in the returned DDX.

```
<Constraint>
    <NoAttributes />
    <Project variable="/sst"/>
    <Select condition="/sst/lat>24.5" target="sst"/>
    <Select condition="/sst/lon<-50.5" target="sst"/>
</Constraint>
```

Example 36:

This constraint expression exercises the *Project Function* `make-sst`.

```
<Constraint>
  <Project function="make-sst(/raw-count, 223)"/>
</Constraint>
```

References

[1] H. Alvestrand. IETF policy on character sets and languages. RFC 2277.

[2] Ken Arnold and James Gosling. *The Java Programming Language*. Addision Wesley, Reading, Massachusetts, 1996.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396.

[4] C.J. Date. *An Introduction to Database Systems*. Addison Wesley, Reading, Massachusetts, 2000.

[5] Oliver Dubuisson. *ASN.1: Communication Between Hetergeneous Systems*. Morgan Kaufmann, San Francisco, California, 2001.

[6] International Organization for Standardization. ISO 8601:2000. http://www.iso.org/, seach for '8601', 2000. Accessed 13 October 2003 on the World Wide Web.

[7] NCSA. HDF 4.1r3 user's guide. http://hdf.ncsa.uiuc.edu/UG41r3_html/, 1999. Retrieved from the World Wide Web 13 October 2003.

[8] NCSA. HDF5 - a new generation of HDF. http://hdf.ncsa.uiuc.edu/HDF5/, 2001. Retrieved from the World Wide Web 15 December 2002.

[9] Russ Rew, Glenn Davis, and Steve Emmerson. *NetCDF User's Guide*. Unidata Program Center, Boulder, Colorado, April 1993. Version 2.3.

[10] Tom Sgouros, James Gallagher, and Peter Cornillon. Place holder for our paper on the dap 3 and syntacitc versus semantic metadata. *Irreproducible Results*, 0(0):0–0, 2004.

[11] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, Massachusetts, 1984.

[12] University of Wellington, Victoria. ISO 8601 date/time representations. http://www.mcs.vuw.ac.nz/technical/software/SGML/doc/iso8601/ISO8601.html, 2001. Retrieved 13 October 2003 from the World Wide Web.

[13] ANSI. C.

[14] ANSI. C++.

[15] IEEE.

A XML Schema

The XML schema has not been fully formalized and will appear here (in this section) at later point in this documents development.

B Error Codes

The error messages and codes issued by a DAP server are shown in Table 8. They are largely taken from the HTTP error codes outlined in the HTTP standard. The code and the title MUST be delivered as shown here. The *description* MAY be altered if you want to translate it into another language.

Table 8: Error Codes

Code	Title	Description
400	Bad Request	The URL could not be resolved. The host name is probably incorrect.
401	Unauthorized	The resource requested is not available without authentication, and yours has failed.
402	Payment Required	
403	Forbidden	
404	Not Found	The data file specified in the request is not on the specified server.
408	Request Timeout	
409	Conflict	
410	Gone	
411	Length Required	
412	Precondition Failed	
413	Request Entity Too Large	
414	Request-URI Too Long	
500	Internal Server Error	
501	Not Implemented	
502	Bad Gateway	
503	Service Unavailable	
504	Gateway Timeout	

C Change log

```
$Log: dap_objects.tex,v $
Revision 1.64  2003/11/03 03:44:27  jimg
Changes plus parallel additions to the TODO document.

Revision 1.63  2003/11/01 00:42:12  ndp
*** empty log message ***

Revision 1.62  2003/10/30 18:56:54  ndp
*** empty log message ***

Revision 1.61  2003/10/30 02:42:49  ndp
*** empty log message ***

Revision 1.60  2003/10/29 22:58:25  ndp
*** empty log message ***

Revision 1.59  2003/10/29 19:16:24  ndp
```

```
*** empty log message ***

Revision 1.58  2003/10/29 01:49:50  ndp
*** empty log message ***

Revision 1.57  2003/10/28 01:39:34  ndp
*** empty log message ***

Revision 1.56  2003/10/28 00:54:33  jimg
check point

Revision 1.55  2003/10/27 15:13:08  ndp
*** empty log message ***

Revision 1.54  2003/10/26 19:41:51  ndp
*** empty log message ***

Revision 1.53  2003/10/24 23:51:25  ndp
*** empty log message ***

<<<<<<< dap_objects.tex
=======
Revision 1.52  2003/10/24 23:21:46  jimg
check point

Revision 1.51  2003/10/24 22:17:52  ndp
*** empty log message ***

>>>>>>> 1.52
Revision 1.50  2003/10/24 21:25:20  ndp
*** empty log message ***

Revision 1.49  2003/10/24 20:14:46  ndp
*** empty log message ***

Revision 1.48  2003/10/24 20:04:37  jimg
check point; some minor, annoying, fixes.

Revision 1.47  2003/10/24 18:34:23  jimg
Fixed a misspelled latex command (hosed and 'end description')

Revision 1.46  2003/10/24 18:29:46  jimg
Added text for the 'information model' of CE.

Revision 1.45  2003/10/23 23:48:08  ndp
*** empty log message ***

Revision 1.44  2003/10/23 23:43:52  ndp
*** empty log message ***

Revision 1.43  2003/10/22 22:55:39  jimg
I've reorganized the Response section (which is still called the Objects
section) so that Version is part of Capabilities. I've moved Client Server
Interaction so that it follows the data model and split the CE section into
two parts, one that's part of the Data model half of the spec and one that's
part of the XML syntax half. There's still tons to do...

Revision 1.42  2003/10/22 02:08:23  jimg
check point
```

```
Revision 1.41  2003/10/21 23:59:17  ndp
*** empty log message ***

Revision 1.40  2003/10/21 23:48:47  ndp
*** empty log message ***

Revision 1.39  2003/10/21 23:30:40  jimg
check point

Revision 1.38  2003/10/21 21:55:00  ndp
*** empty log message ***

Revision 1.37  2003/10/21 20:35:46  ndp
*** empty log message ***

Revision 1.36  2003/10/21 15:24:14  jimg
check point

Revision 1.35  2003/10/17 23:33:52  ndp
*** empty log message ***

Revision 1.34  2003/10/17 00:46:22  ndp
*** empty log message ***

Revision 1.33  2003/10/16 23:56:07  ndp
*** empty log message ***

Revision 1.32  2003/10/16 23:51:45  jimg
Added more detail to the subsection onthe Blob.

Revision 1.31  2003/10/16 21:23:35  ndp
*** empty log message ***

Revision 1.28  2003/10/16 16:45:32  jimg
check point...

Revision 1.27  2003/10/15 23:15:39  jimg
check point

Revision 1.26  2003/10/15 15:26:29  jimg
check point

Revision 1.25  2003/10/15 00:01:03  jimg
check point

Revision 1.24  2003/10/14 22:23:35  jimg
... changes to Section three.

Revision 1.23  2003/10/14 22:21:41  jimg
... the start of some changes to Section three.

Revision 1.22  2003/10/14 00:14:54  jimg
Changes to Sections 1 and 2. I've added the atomic types Enumeration, Boolean
and Time. Also, I moved tom's note about unlimited sizes to the section
about blobs and made it part of the spec. I _think_ I have changed all the
size specifications in Section 2 to say 'the unlimited size thing.'

Revision 1.21  2003/09/23 16:49:38  ndp
```

```
*** empty log message ***

Revision 1.20  2003/09/18 22:56:06  ndp
Added comments...

Revision 1.19  2003/09/18 15:57:01  jimg
Fixed problems which prevented pdflatex from running. It seems pdflatex is
pickier about latex errors than just plain latex...

Revision 1.18  2003/09/17 22:42:24  jimg
Changes. There are a number of issues which need to be resolved; some are
mentioned in the text as Notes, others are parts of the DAPFourSpec topic on
our TWiKi.

Revision 1.17  2003/07/24 22:32:12  tom
excised dap_services document references

Revision 1.16  2003/07/16 04:06:25  tom
incorporated all known change requests

Revision 1.15  2003/07/16 01:06:08  tom
progress on comments, fixed titles

Revision 1.14  2003/06/10 16:13:53  tom
incorporated suggestions from March meeting

Revision 1.13  2003/06/10 01:00:04  tom
CE syntax fixes

Revision 1.12  2003/06/05 21:00:58  tom
changed CE syntax

Revision 1.11  2003/05/31 01:30:18  tom
fixed constraint expression description in dap_objects to be XML

Revision 1.10  2003/05/28 21:06:50  tom
progress 5/28

Revision 1.9  2003/05/23 21:50:52  tom
progress made

Revision 1.8  2003/05/22 19:37:30  tom
rearranging

Revision 1.7  2003/04/10 16:10:21  tom
modifications at opendap meeting

Revision 1.6  2003/03/19 21:48:06  tom
progress made, ready for the March 03 DODS mtg

Revision 1.5  2003/03/17 17:45:10  tom
progress made.  draft for discussion 3/18/03

Revision 1.4  2003/03/13 17:35:15  tom
progress made. not finished

Revision 1.3  2003/03/03 05:34:28  tom
progress entry
```

```
Revision 1.2  2003/02/28 20:59:32  tom
progress made, 2/28

Revision 1.1  2003/01/14 19:55:31  jimg
Added.
```