GWD-I Martin Westhead, EPCC, University of Edinburgh
Category: INFORMATIONAL Alan Chappell, Pacific Northwest National Laboratory
GGF Data Format Description Language Working Group
10th September2003

Data Format Description Language – Structural Description

Status of This Memo

This memo provides information to the Grid community regarding the specification of a Data Format Description Language. The specification is currently an early draft which does not represent a consensus within the group. Distribution is unlimited.

Abstract

XML provides an essential mechanism for transferring data between services in an application and platform neutral format. However it is not well suited to large datasets with repetitive structures, such as large arrays or tables. Furthermore, many legacy systems and valuable data sets exist that do not use the XML format. The aim of this working group is to define an XML-based language, the Data Format Description Language (DFDL), for describing the structure of binary and character encoded (ASCII/Unicode) files and data streams so that their format, structure, and metadata can be exposed. This effort specifically does not aim to create a generic data representation language. Rather, DFDL endeavors to describe existing formats in an actionable manner that makes the data in its current format accessible through generic mechanisms.

DFDL will be represented primarily in XML. This document is a precursor to the definition of the XML representation that establishes a formal semantics for the language.

Contents

1. Introduction

DFDL aims to provide the following:

- a way to describe the structure of a binary sequence (e.g. file or bit stream)
- a way to attach semantic labels to features of that structure
- one or more ontologies of such semantic labels.

This document focuses on the underlying semantics of the first of these goals. Fundamental to DFDL is the concept of a structured binary sequence. The next section provides a formal definition of structured binary sequences and the remainder of the document defines a formal language for describing them.

This formal language will provide the foundation of DFDL. After definition, the language will be represented in XML, defining DFDL syntax. Achieving basic functionality then requires the definition of mechanisms linking the structural descriptions to other forms of computational representation (e.g. a way to link 32 bits, which we have identified as a floating point number, to a routine that can convert those bits into a Java float taking into account whatever representational issues exist.)

Conventions:

- Terms are italicized when being defined.
- Set notation: $\in$ is an element of, $\subseteq$ subset, $\subset$ strict subset, $\equiv$ set equality (A≡B iff A is a subset of B and B is a subset of A), intersection $\cap$ and union $\cup$.
- We use the assignment operator :=
- Set builder notation {x | foo} “the set of things x such that foo”.
- *A-B* for sets *A* and *B* is the set $x \in A\text{-}B$ iff $x \in A$ and $x \notin B$.

2. Structured binary sequence

A *sequence* is an ordered list of objects written [*x*; *y*; *z*] for some objects $x, y, z \in X$ (for some set *X*). Where ‘[‘ and ‘]’ denote the start and end of the sequence and ‘;’ is used to separate the individual elements. The empty sequence is the sequence with no values written [].

The concatenation operator ‘::’ is used to join two sequences thus: []::[*x*] := [*x*], [*x*]::[*y*] := [*x*; *y*], [*x*]::[*y*; *z*] := [*x*; *y*; *z*] and so on.

A *binary value* is an element of the set {0, 1}. A *binary sequence* is a sequence where each of the objects in the sequence are binary values, for example [0; 1; 1; 0; 1; 1]. (N.B. white space is ignored).

We can also write, for example, [B; B; B] where B={0, 1} to denote a set of binary sequences. In this case this is the set: {[0;0;0],[0;0;1],[0;1;0],[0;1;1],[1;0;0],[1;0;1],[1;1;0],[1;1;1]}. Formally we define the operator ‘;’ over sets as:

$$[] := \{[]\}$$
$$[X; Rest] := [X; Rest] = \{ [x]::r \mid \forall x \in X, \forall r \in [Rest] \}$$

The ‘::’ operator can be extended to cover sets as follows:

$$S1::S2 = \{ s1::s2 \mid \forall s1 \in S1, \forall s2 \in S2 \}$$

A *structured binary sequence* is an ordered list containing binary sequences and/or structured binary sequences. It can be written using nested square brackets, e.g.

[[[1;1;0;1];[1;1;0;1]];[1;1;0;1]]. The sequences that make up a structured binary sequence are referred to as its *subsequences*.

For any structured binary sequence there is a uniquely defined *underlying binary sequence*, representing the binary stream without structural groupings. The underlying binary sequence U is constructed from a structured binary sequence S by concatenating the binary values from S in the order in which they occur within the structure, e.g. S = [[[1;1;0;1];[1;1;0;1]];[1;1;0;1]] has the underlying binary sequence U = [1;1;0;1;1;1;0;1;1;1;0;1]. Any binary sequence will have an infinite number of possible structured binary sequences associated with it. E.g. the sequence [0] can be associated with [0], [[0]], [[[0]]], and so on. A structured binary sequence S is said to *describe* a binary sequence *u* iff *u* is the underlying binary sequence of S.

Much of our attention will focus on sets of structured binary sequences which we will refer to as *sequence sets* or *types*, [[X; Y]; [Z; W]] for example. A sequence set *S* can be used to describe a binary sequence *u* if *u* is the underlying binary sequence of a structured sequence *s,* where $s \in S$.

3. Describing simple sequences – repetition and parameters

The aim of this document is to provide a language for succinctly defining sets of structured binary sequences. We begin by formally defining a bit:

bit:= [B], B={0,1}

Then we can build arbitrary types from this. For example we can intuitively think about the sequence set that defines an 8 bit sequence we can call a byte[1].

byte:= [bit; bit; bit; bit; bit; bit; bit; bit]

The first element of the language, then, is a *semantic label* used to name and represent other parts of a description, e.g., byte in the example above. Semantic labels are intended to both describe components and to provide a shorthand notation. As such, semantic labels provide for hierarchic data descriptions as follows:

float := [byte; byte; byte; byte]

is equivalent to

```
float := [[bit; bit; bit; bit; bit; bit; bit; bit],
          [bit; bit; bit; bit; bit; bit; bit; bit],
          [bit; bit; bit; bit; bit; bit; bit; bit],
          [bit; bit; bit; bit; bit; bit; bit; bit]]
```

Note that in order to allow the description of tree structures of arbitrary depth (see example in Appendix 1) the ':=' assignment can be recursive.

The next element of the language will be the repetition operator '.', so we can write:

byte := [bit.8]

[1] N.B. We use the terms float and byte to help the reader understand the context and motivation. It is proposed as a separate exercise to define names and labels for characterising one or more sets of standard names and labels for identifying structures.

Formally we can define '.' as follows:

$$X.n := \begin{cases} \{[]\} & \text{iff } n = 0 \\ X.(n-1)::[X] & \text{iff } n \geq 1 \end{cases}$$

We also allow the notations c.*, c.+, c.? defined as:

$$X.* := \{s \mid \forall n \in Z \text{ s.t. } n \geq 0, \forall s \in X.n\}$$
$$X.+ := \{s \mid \forall n \in Z \text{ s.t. } n \geq 1, \forall s \in X.n\}$$
$$X.? := \{s \mid n=0 \text{ or } 1, \forall s \in X.n\}$$

(where Z is the set of integers).

So now we can define, for example:

float := byte.4
int := byte.4

Notice that float ≡ int. However, the semantic label can then be used to define the way in which the bits should be interpreted.

To allow flexibility, succinctness, and greater comprehensibility for users, the next language element is formal parameterization of semantic labels. For example:

array(type, size) := [type.size]

This parameterisation is very important if we are to allow the sort of abstraction required to allow us to describe high-level data structures whilst leaving the features of the underlying physical representation open.

4. Labels

It is important that we can label the pieces of the structure that we are describing. It is using these labels that we can assign meaning (in terms of the operations that can be performed on the structure) to the structures. We will use two types of labels called names and attributes. We have already introduced examples of names (e.g. int, byte, float, etc.). Attributes are name value pairs that can be used to give additional information about a structure. They are notated using subscripts enclosed in '<>' brackets.[2]

Thus we can label a particular sequence of 32 bits as:

float$_{<byteOrder=bigEndian>}$

or we could define:

myStruct := [float$_{<label=xPosition>}$, float $_{<label=yPosition>}$, float $_{<label=velocity>}$]

or

[2] Originally these had been represented as just footnotes, which is easy to read here but difficult to write down in a text email.

complex := [float <label=real>, float <label=imaginary>]

To emphasise the attachment between the name and the group of labels and to pave the way for referencing the data we allow the data to be written with each structure prefixed by its name and labels. For example:

Suppose we have a binary sequence S which is described by the sequence set array(complex,2) we might write it:

```
array(complex,2)[
complex[float <label=real>[byte [bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]; bit[0]; bit[1]];
                           byte [bit[0]; bit[1]; bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]]
                           byte [bit[0]; bit[0]; bit[0]; bit[1]; bit[1]; bit[1]; bit[1]; bit[0]]
                           byte [bit[1]; bit[0]; bit[1]; bit1]; bit[1]; bit[0]; bit[0]; bit[1]]];
        float <label=imaginary>[byte [bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]; bit[0]; bit[1]];
                           byte [bit[0]; bit[1]; bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]]
                           byte [bit[0]; bit[0]; bit[0]; bit[1]; bit[1]; bit[1]; bit[1]; bit[0]]
                           byte [bit[1]; bit[0]; bit[1]; bit1]; bit[1]; bit[0]; bit[0]; bit[1]]]
    ]
complex[float <label=real>[byte [bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]; bit[0]; bit[1]];
                           byte [bit[0]; bit[1]; bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]]
                           byte [bit[0]; bit[0]; bit[0]; bit[1]; bit[1]; bit[1]; bit[1]; bit[0]]
                           byte [bit[1]; bit[0]; bit[1]; bit1]; bit[1]; bit[0]; bit[0]; bit[1]]];
        float <label=imaginary>[byte [bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]; bit[0]; bit[1]];
                           byte [bit[0]; bit[1]; bit[0]; bit[0]; bit[1]; bit[1]; bit[0]; bit[1]]
                           byte [bit[0]; bit[0]; bit[0]; bit[1]; bit[1]; bit[1]; bit[1]; bit[0]]
                           byte [bit[1]; bit[0]; bit[1]; bit1]; bit[1]; bit[0]; bit[0]; bit[1]]]]
        ]
]
```

For convenience we might shorten this to:

```
array(2, complex)[
   complex[float <label=real>α('1.002'); float <label=imaginary>α('0.1')];
   complex[float <label=real>α('2.034'); float <label=imaginary>α('0.4')]
]
```

where 'α()' is a function that maps from our representation of a value, in this case a string representation of a floating point number, to that would describe it. We will assume that a suitable function exists. For convenience we will omit the α() and where the meaning is clear we will simply enclose sequences of characters in square bracketed quotes (e.g. ['1.002']) to denote the binary sequence set given by its alpha mapping (e.g. α('1.002')).

So, the calculus so far allows the description of C-struct-like structures (fixed sequences) multi-dimensional arrays (as arrays of arrays), tables (arrays of structs), and sequences of the above. Furthermore we can label these objects with arbitrary labels.

4.1 Attribute groups and scope

In order to keep the representations from getting too verbose we introduce two pieces of "syntactic sugar" to the writing of attributes:

1. attribute groups
2. hierarchical scope

4.1.1 Attribute groups

If we have several attributes that may be applied to separate sequence sets (perhaps semantics associated with the physical source of the data sequence), we can group and label these properties together and apply that group through use of the label:

localStoreGroup := <byteOrder = bigEndian, src = “http://www.myhost.org/data/localstore”, lastUpdated = “2/5/03”,…>

array(2,complex) <localStoreGroup>

Attribute groups may be created hierarchically and parameterized, similar to semantic labels:

encodedLocalStoreGroup(storeEncoding) := <localStoreGroup, encoding=”storeEncoding”>

4.1.2 Hierarchical Scope

When an attribute with a name value pair <*N* = *V*> is applied to a particular sequence *S*, they automatically apply to that sequence *and* each of its subsequences *S', unless S'* explicitly defines a label *N* = *V'* in which case the value *V'* takes precedence.

So:
array(2, complex) <myValue=1> [
  complex[float [‘1.002’]; float[‘0.1’]];
  complex[float[‘2.034’]; float<myValue=4000>[‘0.4’]]
]

Is equivalent to:
array(2, complex) <myValue=1> [
  complex<myValue=1>[float<myValue=1>[‘1.002’]; float<myValue=1>[‘0.1’];
  complex<myValue=1>[float<myValue=1>[‘2.034’]; float <myValue=4000>[‘0.4’]
]

ToDo: Need to add indexing labels for arrays

5. Delimited sequences

There are many examples of files in which the length of structural sequences is determined by the use of a terminating value, for example null-terminated strings or comma separated values. In fact we already have sufficient notation to do this. For example a comma separated table might be represented by:

```
valueChar   :=  char - ( lineFeed | comma )
field       :=  [ (valueChar.* ); comma]
finalField  :=  [valueChar; lineFeed]
row         :=  ( (field.* )::[finalField] )
table       :=  ( row.* )
```

In this example, a “valueChar” is character that does not contain a comma or a line feed. A “field” is a sequence of a “valueChar”s followed by a comma. A “finalField” is like a field except the terminating character is a lineFeed return. A “row” is a sequence of zero or more “fields” followed by a “finalField” and a table is a sequence of zero or more “rows”.

The definitions of “comma”, “lineFeed” and “char” are assumed. These are actually defined in the strawman primitives ontology.

Notice that in this example we would like:

1. the “field” character to represent the data itself *without* the comma and

2. we need to be able to distinguish between comma and lineFeed in order to recognise the end of a row.

In order to do this we have to exclude these terminating characters from the sequence set "field"

In other situations we would like to include the termination character, such as null terminated strings. This can be done by explicitly including the extra character:

string := [(char-['\0']).*] :: ['\0']

6. Data references

Being able to reference the data is very powerful. It can enable:

1. Conditional structures, such as unions where different structures are applied depending on the value of part of the data.
2. Variable sized arrays where the array size is given as part of the data itself.
3. Simple integration – the ability to describe a data set composed of pieces of data from other sequences.
4. Simple transformation – selection and rearrangement of pieces of data.
5. Data annotation

We can use the names and semantic labels in a sequence set *S* that describes a binary sequence *b* to define a path that can refer to a portion of *b* described by a substructure within *S*. For example the path: "/array/ complex <indexValue=2>/float <label=real> " would be used to refer to the real part of the second element of the complex array defined earlier. We will not provide a formal definition of the path here, since in practice XPath is probably the mechanism that would be used to define it, within an XML description. However we will assume that an unambiguous path can be constructed to any subsequence within the structured sequence. In the case where multiple sequences result from a single path the first is assumed, and the notation "." is used to refer to the local sequence.

6.1 Variable sized arrays

For a variable sized array we need to have a way to convert from a bit sequence to an integer. Let us assume that such a mapping exists and call it β(). Then we can write:

size := int
variableArray(*type*) := [size; array(type, β("../size"))]

7. Conditionals

DFDL supports two kinds of conditional:

1. Conditional based on data reference
2. Conditional based on pattern.

7.1 Data reference

Using the data reference we can provide a choice operator:

[(*path,*Y_1:X_1, Y_2:X_2,…)]:= X_n

Where *path* is a data reference path as described above and Y_i and X_i are sequence sets such that:

$$Y_i \cap Y_{i'} = \varnothing, \forall i \neq i'$$

And *path* refers to a binary sequence *b* such that *b* can be described by Y_n.

For example:

myUnion := [int; (“..\int”, α(0):float, α(1):complex)]

Here myUnion is a conditional data structure, it always starts with an int, if that int is 0 (notice that we use the α() mapping again to map from values to bit sequences) then it is followed by a float. If the int is 1, it is followed by a complex.

7.2 Pattern Matching

In analogy to the alternation character in regular expressions, we will use ‘|’ as an alternative notation for the union operator ‘∪’ This simply expresses alternative patterns in the sequence set. It is primarily expected to be used for text matching. For example:

[(‘a’ | ‘b’); ‘x’]

Is the set of structured sequences that represent the sequences [‘a’; ‘x’] and [‘b’; ‘x’]. Note, the ‘()’ brackets are just used for grouping.

8. Pointers within the data

The data references in the previous section enable us to refer to structurally significant sequences in the underlying binary sequence through its description. However, there will also be situations in which there is a need to reference one piece of the data from another piece. We will call this sort of reference a pointer to distinguish from the type of references used in the earlier section.

Examples of pointers will be found in files that describe complex relationships between objects, such as graphs (meshes/grids etc.) or highly structured databases.

ToDo: The details of pointers are under discussion

9. Transformations

Within DFDL we need to be able to describe transformations that may occur on the digital entity we are describing. Such transformations include:

1. Compression (e.g. zip, gzip, run length encoding)
2. Encryption
3. ASCII encoding (e.g. binhex, uuencoding etc.)
4. Techniques to improve disk efficiency (alignment/blocking, striping etc.)

Many such transformations will perturb the structure so that the binary structure to which the DFDL description refers. Thus we distinguish between the *physical binary sequence* which refers to the bits on the disk and the *conceptual binary sequence* which refers to the sequence described by a piece of DFDL. Note that there may be many transformations in between a physical representation and a conceptual one, and there may be multiple conceptual representations in a complex data path.

Whilst DFDL may be able to capture the semantics of some simple transformations (subsetting and reordering), we want to characterize arbitrary binary transformations. So we take the approach that within the structural description language a transformation can be labeled, and the structure of its input and output described. The semantics of the transformation are described in an appropriate ontology.

Supposing we had a file containing a CSV table (csvTable) that had been compressed we would write:

compressed(~csvTable~)

(The modified brackets '(~', '~)' designate the extent of the transformation).

Additionally attributes can be added to the transformation, for example to define the algorithm used

Compressed<algorithm="runLength">(~csvTable~)

ToDo: this needs some work: Is this the right representation for a transformed file? Is it possible to link transformations that are reversible (compress and uncompress)?

We also need to be able to describe partially transformed files (e.g. compressed region or digital signature). This is tougher though since we may have to say things about the transformation before and also after it is transformed in such a way that we do not confuse one with the other,

10. Appendices

10.1 Appendix 1 – XML example

In this appendix we provide an example DFDL description of the syntax of XML. The example is intended to illustrate the power and elegance of the language. It is not intended to be a complete description for XML

We begin by defining the following character sets:

- *c* is the sequence set representing legal XML value characters (i.e. no '<', '>', '=', '&' etc.)
- *w* is the sequence set representing whitespace characters

xmlDocument := matchedTag

matchedTag : = [openTag; content; matchedTagSequence; content; closeTag]

openTag := ['<'; tagName; attributeSequence; '>']

content := [*c*.*]

matchedTagSequence := matchedTag.*

closeTag := ['<'; "../openTag/tagName"; '>']

attributeSequence := [whitespaceSequence; attribute].*

attribute := [attributeName; '='; attributeValue]

tagName := [*c-w.**]

attributeName := [*c-w.*]*

attributeValue := [*c.**]

whiteSpaceSequence := [*w*.*]

10.2 Outstanding issues

This is a draft document. This appendix provides a list of the current issues at time of writing:

- Index label – we would like to define an automatic labeling with respect to sequence number, particularly for arrays. It is not clear how best to represent it right now.
- Pointers – the details of data references to other points in the file are under discussion.

11. Security Considerations

There are no security considerations that we are aware of at this time.

Author Information

Martin Westhead, M.Westhead@epcc.ed.ac.uk, EPCC, University of Edinburgh. James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK.

Alan R. Chappell, chappella@battelle.org, Pacific Northwest National Laboratory, Battelle Seattle Research Center, 4500 Sand Point Way NE, Suite 100, Seattle, WA 98105-3949

Glossary

DFDL – Data Format Description Language

Intellectual Property Statement

Full Copyright Notice

References

BinX http://www.epcc.ed.ac.uk/gridserve/WP5/Binx/
HDF http://hdf.ncsa.uiuc.edu/HDF5
BDF/SAM http://collaboratory.emsl.pnl.gov/docs/collab/sam
XDR http://www.faqs.org/rfcs/rfc1014.html
DFDL web pages http://www.epcc.ed.ac.uk/dfdl