Mike Beckerle, Ascential Software

2004-03-23

DFDL

Proposal & Examples

(DRAFT)

DFDL: Proposal and Examples

(Draft)

Status of This Memo

Copyright Notice

Abstract

A Data Format Description Language (DFDL) is proposed as an open industry standard for describing all kinds of data. Rather than prescribing a data format, DFDL is descriptive of existing formats including the handling of important commercial data processing requirements for record-oriented data. It also handles scientific data processing requirements such as multi-dimensional arrays. Important legacy data formats are covered (e.g., like packed decimal numbers, VBS blocking formats)

Revision History

Latest entry ***at the top*** please			
Version	Author/Contributor	History	Date(yyyy-mm-dd)
008	Mike Beckerle	Stripped huge amounts of details out. Leaving just central stuff. Working draft, NOT FOR DISTRIBUTION.	2004-03-21
007	Mike Beckerle	First public draft for DFDL-WG member commentary and feedback. Formated to GGF standards based on Ascential Internal version 006. There remains a "To Do" list at the end of the document and many "TBD" items within.	2003-09-22

Contents

1. Introduction

This is a proposal for a DFDL with the following characteristics

- Leverage XML technologies such as XSD.
- Keep simple things simple
- Allow adequate expressive power for more complex situations

This proposal is related to earlier drafts, and also do the document "DFDL XML Approach" by Martin Westhead, circulated 2004-01-30 on the GGF DFDL Working group mailing list.

1.1 Glossary

DFDL – Data Format Description Language

The term “byte” herein refers to an 8-bit octet.

Herein where the phrase "must be consistent with" is used, it is assumed that a conforming DFDL implementation must check for the consistency and issue appropriate diagnostic messages.

(TBD: uniform way to talk about suppressibility of warnings/errors?)

A “DFDL processor” is a program that uses DFDL descriptors in order to process data described by them.

2. Proposal

Like XSD, data is described as elements which can be composed of other elements or can be "leaf" types. These can be built-in types, or can be defined by a type-definition mechanism.

Elements have name, type, dimension, and nillability. To this we add representation or "rep" for short.

This association of element definition to rep definition can be directly, i.e., adjacent in the text, or indirectly from another place in the file or another file using references to names/paths.

The rep information is ultimately information on how to transform the rep into the value space for input and value space to rep for output. The specification of the transform is accomplished by specifying 3 kinds of rep information:

- what: what is the rep information, in other words, what is the type of the rep?
- how: how do I interpret the rep information to compute a value? For output it's how do I interpret the value information to compute the rep?
- where: where is the rep information located?

So, an element definition with every aspect of rep expressed directly looks like:

```
<element name="name" ...>
  <...type definition ...>
  <rep ...>  // where info is in the attributes of the rep
    <repType ...>... // what IS the rep
    </repType>
    <valueCalc .../> // How do I convert the rep to a value
    <repCalc .../>   // How do I convert the value to a rep
  </rep>
</element>
```

This proposal has some real differences but also strong relationships to the most recent proposal by M. Westhead (delivered 2004-01-30 to the dfdl-wg mailing list)

Instead of source and transform we have container and valueCalc for interpreting source data. Container and repCalc perform the opposite function, calculating output or target data representation. The notion of container generalizes source and target without specifying direction of data movement or interpretation. The symmetric pair of valueCalc and repCalc provide a place to put the expressions which compute both the abstract data value from the rep and vice versa. If the valueCalc is invertible it may be possible to leave the repCalc unspecified and have it computed automatically as the inverse of the valueCalc.

Our notion of container is quite a powerful generalization because a Container is an Element, and so it recursively can have a type and rep of its own. This is used in situations like the Matrix example where we have to express multiple layers of preprocessing:

```
<container name="charStream" type="string">
  <rep charset="UTF-8"
       container="byteStream"> <!-- a built in container -->
    <valueCalc exp="{ bytesToChars() }"/>
             <!-- TBD: reconcile with length computation.... -->
  </rep>
</container>

<container name="noCommentsStream" type="string">
  <rep container="charStream">
    <valueCalc exp="{replaceString( '...a regexp for comments...', ' ')}"/>
  </rep>
</container>

<container name="noBlankLinesStream" type="string">
  <rep container="noCommentsStream">
    <valueCalc exp="{ replaceString( '..a regexp for blanklines..',' ')}"/>
  </rep>
</container>
```

These three derivations each perform preprocessing on their input "container" and construct a "container" which encloses the rep for the next tier of interpretation. There is a built in container whose name is "byteStream" here. (Perhaps should be renamed to avoid the "stream" issue. Maybe just "bytes" is the right name, or byteContents or byteContainer?)

(TBD: are containers where we do lazy evaluation?)

We'll see how these container definitions are used more fully in the complete Matrix example in a later section.

We allow multi-step transformations by the fact that the rep of something can itself be described as an element. I.e., using the abstract model. This gives us multi-layered transforms. This is used in the clever string example:

```
<element name="addr" type="string">
  <rep>
    <element name="cleverString"> <!-- recursively rep is an element -->
      <sequence>
        <element name="lengthField">...</element> <!-- this has its own rep! -->
        <element name="contentField">...</element> <!-- this has its own rep! -->
      </sequence>
    </element>
  </rep>
</element>
```

In this fragment we show that the rep contained within the "addr" element is defined in terms of an element which is a sequence containing a length and content field each of which can have their own complex representation.

We'll see this example again later and more completely.

Having looked at the flavor of the proposal briefly, now let's look at some examples one by one.

3. Example 1: Simple record of 3 numeric fields

We'll study four variations on this example. All of them are of a file or data feed containing exactly 3 numbers:

1. X: int;
2. Y: float;
3. Z: double;

The data is stored in big-endian byte order in a binary file.

3.1 Example 1A: XSD + separated representation information

The purpose of this example is to show how this proposal keeps simple things simple, and how we leverage XSD.

```
<element name="example1">
  <sequence>
    <element name="x" type="int"/>
    <element name="y" type="float"/>
    <element name="z" type="double"/>
  </sequence>
</element>

<!-- here we define a rep in a named reusable fashion -->
<rep name="bigEndianBinaryRep"
     byteOrder="bigEndian"/>

<!-- here's how we associate a rep with an element after the fact -->
<element ref="example1">
  <rep ref="bigEndianBinaryRep"/>
</element>
```

In the above we see that we have an ordinary XSD definition for the data, followed by a separate non-XSD declaration of a reusable named representation or "rep" for short. This is followed by a separate statement matching the particular named element with a rep.

This version lets all kinds of potentially specified variations all take on default values. We'll elaborate what most of these are in a subsequent example.

3.2 Example 1B: XSD + Embedded representation information

A similar example but here we haven't separated the rep information, we've embedded it directly.

```
<element name="example1">
  <rep byteOrder="bigEndian" />
  <sequence>
    <element name="x" type="int"/>
    <element name="y" type="float"/>
    <element name="z" type="double"/>
  </sequence>
</element>
```

This saves a few characters of typing and increases density, but is less flexible.

Note that if you want "element" here to really be "xsd:element" and this to be a real XML schema, then you must encapsulate the "rep" construct in "xs:annotation" and "xs:appinfo" constructs, but otherwise the representation would be the same.

3.3 Example 1C: XSD + Embedded representation, Explicit information (no Defaults)

This example is the same simple structure, but we've elaborated many things that could be omitted as they are the default choices.

```
<element name="example1">
  <sequence>
    <element name="x" type="int">
      <rep
        byteOrder="bigEndian"
        position="0"
        container="byteStream">
        <valueCalc exp="{ (bytesToInt(), 4) }"/>
        <repCalc exp="{ (intToBytes(), 4) }"/>
      </rep>
    </element>
    <element name="y" type="float">
      <rep
        byteOrder="bigEndian"
        offset="0"
        after="../x">
        <valueCalc exp="{ (bytesToFloat(), 4) }"/>
        <repCalc exp="{ (floatToBytes(), 4) }"/>
      </rep>
    </element>
    <element name="z" type="double">
      <rep
        byteOrder="bigEndian"
        offset="0"
        after="../y">
        <valueCalc exp="{ (bytesToDouble(), 8) }"/>
        <repCalc exp="{ (doubleToBytes(), 8) }"/>
      </rep>
    </element>
  </sequence>
</element>
```

Looking at the above XML, we see how each of our 3 contained elements X, Y, and X, has a rep enclosed within its definition. Let's examine the rep for X, and consider the 3 things that the rep information must convey: what, where, and how.

The rep for X specifies the byteOrder attribute. This is type information about the rep. I.e., what is the rep. The where information is in the position and container attributes. The "how" information is in the valueCalc object and the repCalc objects. The valueCalc object contains an expression in an Xquery-like syntax. This expression produces two results: value, and length. Length is measured in the number of elements of the rep that are "consumed" by this value calculation. The function returning two values allows for some functions that compute the value and length. In the example above the length is a known constant for each type, but this isn't true in general. Note that all valueCalc and repCalc transformations are parameterized by all the location and rep properties. The repCalc expression is one which also produces two results: representation contents, and length. In this case the Length is measured in the number of elements of the rep that are created by the calculation. Again this is done this way so that some functions can compute the rep contents and length and return both.

Looking at the rep information for field Y we see that it adds an additional attribute "after". This attribute contains a relative Xpath expression indicating what object this one is found after. When location is specified using the "after" attribute then there is no need to specify the container as the container is assumed to be the same as the container for the rep of the object specified in the Xpath expression. The after attribute is used to make explicit the order of peer elements within a common rep container. It allows the logical order of fields in the abstract type to be different from the order of the fields in the representation.

3.4 Example 1D: XSD + Separated representation, Explicit information (no Defaults)

This variation is similar to the above, but keeps the rep information separated from the XSD of the abstract data.

```
<element name="example1">
  <sequence>
    <element name="x" type="int"/>
    <element name="y" type="float"/>
    <element name="z" type="double">
  </sequence>
</element>

<rep name="bigEndianInt"
  byteOrder="bigEndian"
  position="0"
  container="byteStream">
  <valueCalc exp="{ (bytesToInt(), 4) }"/>
  <repCalc exp="{ (intToBytes(), 4) }"/>
</rep>

<rep name="bigEndianFloat"
  byteOrder="bigEndian"
  offset="0"
  after="../x">
  <valueCalc exp="{ (bytesToFloat(), 4) }"/>
  <repCalc exp="{ (floatToBytes(), 4) }"/>
</rep>

<rep name="bigEndianDouble"
  byteOrder="bigEndian"
  offset="0"
  after="../y">
  <valueCalc exp="{ (bytesToDouble(), 8) }"/>
  <repCalc exp="{ (doubleToBytes(), 8) }"/>
</rep>

<!-- here's where we tie things together -->

<element ref="example1b/x">
  <rep ref="bigEndianInt"/>
</element>
<element ref="example1b/y">
  <rep ref="bigEndianFloat"/>
</element>
<element ref="example1b/z">
  <rep ref="bigEndianDouble"/>
</element>
```

This example is interesting only in that it illustrates that representations can be created and named for reuse, and then element definitions which do not include rep information can be extended to express it. Furthermore, even element definitions which already have some representation information can sometimes be made more explicit by way of further extension.

4. Example 2: Matrix with dynamic size. Rich character representation

In this example we have textual data forming a 2d matrix:

- blank lines are ignored
- C-style comments are ignored (equiv. to whitespace)
- First line contains xdim ydim (whitespace separated)
- Subsequent lines are rows of the 2-d matrix.
- There must be exactly xdim rows, each containing ydim
- numbers.
- Within each row the values are whitespace separated.

- The charset is UTF-8

This example is challenging because it requires that we express preprocessing of the input data to handle the C-style comments and blank lines in a way that is not part of the structure of the data. We begin with the abstract type definition for the matrix:

```
<element name="example2">
<sequence>
  <element name="dims">
    <sequence>
      <element name="xdim" type="int"/
      <element name="ydim" type="int"/>
    </sequence>
  </element>
  <element name="data" type="double">
    <!-- here's a departure from XSD model. XSD has no "array" -->
    <dimension minOccurs="{ $../dims/xdim }" maxOccurs="{ $../dims/xdim }"/>
    <!-- another departure from XSD is that we allow values of one part of the data
         to constrain attributes of other parts of the data -->
    <dimension minOccurs="{ $../dims/ydim }" maxOccurs="{ $../dims/ydim }"/>
  </element>
</sequence>
```

In the above there's a clear departure from the XSD model in that we allow multiple dimensions of an element. Another departure is that we allow values of one part of the data to constrain attributes of other parts of the data. That is, we are allowing the run-time values of xdim and ydim to be used to constrain the min/max occurrences of the dimensions of the matrix.

An alternative to this would be for the expressions to simply say that the metadata indicates that each dimension has minimum zero and maximum unbounded and then use a run-time validation to insure that they match the run-time values of xdim and ydim.

Now we'll look at it with the representation information. Note that valueCalc and repCalc are omitted since they're just the default of "make my value from my rep, with all the other stuff as the parameters" (and the inverse for repCalc).

First we look at how we do the stream preprocessing. This is done via containers defined in terms of other containers.

```
<container name="charStream" type="string">
  <rep charset="UTF-8"
       container="byteStream"> <!-- a built in container -->
    <valueCalc exp="{ bytesToChars() }"/>
                      <!-- TBD: reconcile with length computation.... -->
  </rep>
</container>

<container name="noCommentsStream" type="string">
  <rep container="charStream">
    <valueCalc exp="{replaceString( '/\*(.|\n|\r)*\*/', '&#20;')}"/>
       <!-- regexp is for C-style comments. &#20; is a space. -->
  </rep>
</container>

<container name="noBlankLinesStream" type="string">
  <rep container="noCommentsStream">
    <valueCalc exp="{ replaceString( '(\n|\r\n)(&#20;|\t)*(\n|\r\n)','\n')}"/>
       <!-- regexp is for blank lines, which can contain whitespace characters -->
  </rep>
</container>
```

In the above we assume a built-in container named "byteStream", and define a character stream in terms of it, and then a stream where C-style comments have been eliminated in terms of that, and finally a stream where blank-lines have been eliminated in terms of the previous.

All these streams are of potentially unbounded length, and that is why we need containers to be distinct from elements. When containers are defined in terms of each other we want the design to admit an implementation which avoids bringing the entire representation into memory. By distinguishing containers and the expressions used to compute them we allow for a lazy implementation which will actually compute the representation of a container only as data needs to be drawn from it.

The next part of the example defines the actual representation of the abstract data. Note the rich set of properties associated with textual number parsing. These properties would take on default values and go unspecified in most real-world cases; however, these attributes are shown here so as to make it clear the degree of control available. The other thing we're illustrating is how we can establish the environment for parsing all the numbers in the file with a set of properties at the top-level object of the file. These properties will apply by default to all elements contained within; hence, we do not need to repetitively specify all these representation properties.

The entire rep is defined to come from the container "noBlankLinesStream" above, which is a suitably preprocessed stream of character data that the rep deals only with the layout of the data structure onto it.

```
<element name="example2">
 <rep
  charset="UTF-8"
  fillChar=" "
  isZeroWhenBlank="false"
  signPlacement="prefix"
  digitGroupingSeparator=","
  decimalSeparator="."
  digitGroupingScheme="3"
  generateDigitGroupingSeparator="false"
  exponentCharacters="DdEeFfGg"
  container="noBlankLinesStream">
 </rep>
 <sequence>
  <element name="dims">
    <sequence>
      <element name="xdim" type="int">
        <rep container="dims"/> <!-- contained by parent rep is the default -->
      </element>
      <element name="ydim" type="int">
        <rep
          container="dims" after="../xdim"/> <!-- after prior peer is the default -->
      </element>
    </sequence>
    <rep
     separator="(&#20;|\t)*" <!-- whitespace -->
     container="example2"
     terminator="(\n|\r\n)" /> <!-- newline -->
  </element>
  <element name="data" type="double">
    <dimension minOccurs="{../dims/xdim}" maxOccurs="{../dims/xdim}"/>
    <dimension minOccurs="{../dims/ydim}" maxOccurs="{../dims/ydim}"/>
    <rep after="../dims"
         arrayStorageOrder="rowMajor">
       <dimensionRep separator="(\n|\r\n)" />
       <dimensionRep separator="(&#20;|\t)*" />
     </rep>
  </element>
</sequence>
</element>
```

In the above, the only really interesting thing going on is the separation of the xdim and ydim fields by whitespace and termination of their containing record by end-of-line. The parsing of the 2d array itself is similar. Each dimension specifies a separator.

For multi-dimensional arrays, there is the question of whether the data is stored column or row-major. If this is switched, then the separators in the dimensions must change also.

TBD: rather than giving a regexp for separators and terminators, if what's being expressed is a set of alternatives, then if we provide a way to distinguish one of the alternatives then we are unambiguous w.r.t. what we want done for output. Hence, there needs to be a way to specify multiple separators and multiple terminators as if they were combined into a regexp as alternatives, but the distinguished first one is used for output.

5. Example 3: Clever String

In this example, we have a very simple data type, string, with a somewhat clever representation that is designed for efficiency and space conservation.

The representation of the string is a group of two fields, a length, followed by the string content, encoded as UTF-8 characters. The length gives the length in character units, not in bytes.

The length itself is encoded as follows. If the length of the content is 127 or less then a single byte is used to store the length, with the most significant bit turned on. If the length is greater than 127, then a full 4 bytes is used to store the length. The 4-bytes are not required to be aligned on any particular boundary. Only 31 of the available 32 bits can be used however, since the most significant bit of the first byte must be zero. The integer is big-endian byte order.

This example illustrates the power of the recursion between type and rep. In particular we're going to exploit the fact that the rep of even a simple type can be described by a complex data type, which itself has a complex rep. We begin with the abstract type, which is just an element of type string with a named representation type of "cleverstring":

```
<element name="myString" type="string">
  <rep>
    <element ref="cleverString"/>
  </rep>
</element>
```

Next we build up the representation types. A clever string is a stored length field, followed by the string contents:

```
<element name="cleverString">
  <sequence>
    <element name="lengthField">
       <rep>
         <element ref="cleverLength" />
       </rep>
    </element>
    <element name="contentField" type="string">
       <rep charset="ebcdic-cp-us"
            after="../lengthField"> <!-- the default -->
         <valueCalc exp="{ stringFromBytes($../lengthField) }"/>

       </rep>
    </element>
  </sequence>
  <rep>
    <valueCalc exp="{ ($./contentField, $./lengthField) }" />
  </rep>
</element>
```

The valueCalc expressions are of particular interest in the above. The stringFromBytes takes a number of characters and 'returns' the character codes along with returning the number of bytes consumed in parsing that many characters. It's important to admit an efficient implementation where this doesn't copy the characters.

Finally, we get into the details of how the length is encoded, which is where the cleverness appears:

```
<element name="cleverLength">
  <choice>
     <element name="byteLength">
       <rep>
```

```
          <valueCalc exp="{ (($. && 0x7f), 1) }"/>
          <repCalc exp="{ (($. || 0x80), 1) }" />
        </rep>
      </element>
      <element name="intLength">
        <rep byteOrder="bigEndian"/>
      </element>
  </choice>
  <rep>
    <choiceRep discriminator="{if (($../byteLength && 0x80) = 0x80) then 'byteLength'
else 'intLength' }"/>
  </rep>
</element>
```

The "$." (dollar sign followed by a period) in the path means "where I start in the rep" in both directions. If importing data, then "$." refers to the current item being parsed i.e., "me" in the rep. If exporting data then "$." refers to the current item being exported, i.e., the value of this overall entity which is being converted into an external rep.

Notice that the four byte case has no encoding wierdness in the bits to implement in the calcuations of the length. The basic int type is assumed built-in.

A final point on this example is that we illustrate here a useful breakup of the complexity of this rep into reasonable sized chunks that aren't individually too complex.

6. Example 4: Comma-Separated Values

This example requires some interpretation. When people say the format is "CSV", what's left ambiguous usually is:

1. is there a line of field names as the first line?
2. if not, is the same number of fields required to be on each line? (i.e., this is a table or not?.
3. if the data is NOT all numbers, that is, if some of the data is strings, can the strings ever contain commas? If not ok. If so, then how are these value-internal commas escaped?

We'll assume:

1. yes, field names are first, you have to skip these.
2. each line must contain exactly the number of fields found in the first line
3. values are all strings. If a comma is embedded then the string must be quoted with double quote characters. If the string contains double quote characters then those must be backslash escaped, and backslashes are doubled to escape themselves.

We begin by defining an element called a csvLine, which is used in the definition of a CSV file.

```
<element name="csvLine">
    <sequence> <!--tbd: do we need this sequence? Just one element inside. -->
       <element name="field" type="string" minOccurs="1" maxOccurs="unbounded" />
     </sequence>
     <rep
      separator=","
      terminator="(\n|\r\n)"
           <!-- tbd: does or doesn't quoting scheme apply to terminator too? -->
      charset="iso-8859-1">
        <quoteScheme kind="surrounding" open='"' close='"'
                     generateQuotes="whenNeeded">
            <quoteScheme kind="escape" escapeChar="\">
               <quoteScheme kind="escape" escapeChar="\" />
            </quoteScheme>
        </quoteScheme>
    </rep>
</element>
```

The csvLine above is just a sequence of elements, each a string. These are separated by commas, and the entire csvLine is terminated by end of line. The specification of the quoting sequence is actually the most tedious aspect of this format specification. A specified the quoting sequence doesn't confine itself to just the comma separators; hence, you can include line endings inside a field if you quote the field. This may be undesirable, and it points out that there needs to be a way to tightly bind the specification of a quoteScheme to exactly one or the other of the separator or terminator.

Given the csvLine definition, we can define a CSV file:

```
<element name="csvfile">
 <sequence>
   <element name="headings" ref="csvLine"/>
   <element name="data" minOccurs="0" maxOccurs="unbounded" ref="csvLine">
      <validation exp="{ $../headings/runTimeLength = $./runTimeLength }"/>
   </element>
 </sequence>
</element>
```

In this definition we parse the heading line first, and then the data lines. We require the length of each of the data lines to match that of the heading line using the validation expression.

(TBD: is this the right way to ask for the run-time length?)

To be continued...