A Proposal for a DFDL that handles Commercial Data Processing Requirements

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)
0.7	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. 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. Introduction

There is a broad area of computing that is commonly referred to as *data processing*. In this area data almost always looks like large collections of structured information units called *records*. This data can be in program to program streams, in files (single or multiple), or in databases of various kinds including relational databases. Records vary in structure. A simple record is rows of fields typified by the result set returned from a SQL query on a relational database. More complex data can have records that are hierarchical with nested repeating subarrays of subrecords. Complex records sometimes originate in Cobol programs since Cobol provides extensive support for use of hierarchical data. XML also offers complex hierarchical expressive power, but in contrast to XML, records used in high-volume data processing are represented in densely-packed binary data formats which minimize data size and the amount of calculation required to access the data.

The dominance of record-oriented data in the commercial data processing applications distinguishes it strongly from most scienntific computing and other areas of HPC. Record-orientation motivates its own set of requirements on a data format description. Our hope is that a single DFDL subsumes these requirements along with those of scientific and HPC to create a specification that can serve all Grid computing consituencies.

3. A Vision: The Future of Data

We hope for 3 dominant kinds of file-oriented data in the future:

- Textual XML data
- Binary encoded XML data (see http://www.w3.org/2003/07/binary-xml-cfp.html)
- Data with standardized DFDL descriptors

Textual XML data is by definition new, that is, created in the XML era. In addition it is usually small data because of the large overhead that XML tagging imposes. Binary XML data is also by definition new, and is more reasonable for larger data because of the reduced costs of encoding and more compact size. Both are prescriptive in that they prescribe a representation of the data.

Data with DFDL descriptors represents all the rest of the world’s data. This is the collapsing of every old data format ever invented into a single category based on a DFDL that is capable of describing anything. That is, DFDL is not a format for data. It is a way of describing any data format. Data with DFDL can be old data files for which DFDL descriptors have been written, or it can be new data produced by old programs which have not been updated to create more modern XML-based formats. DFDL can also be crucial for writing new programs which must interoperate with older programs. But the utility of DFDL does not end with accomodation for legacy schemes. DFDL is generally about high-performance and the data representation techniques that come along with it:

- Density: fewest bytes to represent information content (without resorting to compression). Fastest possible I/O.
- Alignment: to insure access to data content with the smallest number of machine cycles

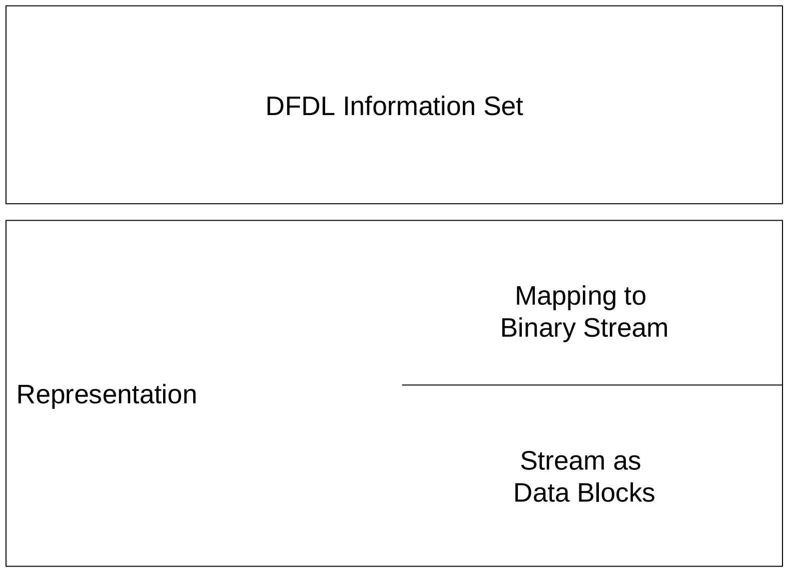
- Memory-mapped I/O: Some DFDL representations will be suitable for processing by directly mapping the binary data into the memory of a process. Use of DFDL insures that this data remains usable outside of this kind of program and that the format is robustly documented and understood.
- Seekable, Random Access: DFDL representations can support seeking to meaningful calculated file offsets without having to read the intervening data.
- Generality: DFDL can describe anything. We insure this by including an expression language within DFDL which is turing-complete. That is, it is capable of expressing real calculations of unlimited complexity. However, it is the spirit of DFDL to make expression of the commonly-found data formats straightforward.
- Read/Write Symmetry. DFDL should specify data representations well enough to allow not only correct interpretation, when data is being read, but an unambiguous specification of how to create data when writing it out.

3.1 About DFDL in XML

DFDL should be an XML-based language. That is, DFDL should be well-behaved XML with a published XML Schema. There are a variety of XML-based languages which were defined in such a way that they cannot have XML Schemas defined. (The ANT build system language for Java programming is an example of this.) We want DFDL to be more ordinary than this. It should have a regular XML schema. This rules out certain syntactic features such as having an extensible set of XML elements that are added to by user type definitions.

4. DFDL Layers

DFDL is best understood by keeping in mind that there are 2 independent layers:



The top layer is the information set. In this layer we are dealing with data independent of its representation. The lower layer is the mapping of that information content to/from concrete representation. This layer actually has two sublayers. The first mapping is from each information item into its representation in an abstract binary stream. Optionally there is a second mapping of the way the stream is represented as blocked data items in the actual physical storage.

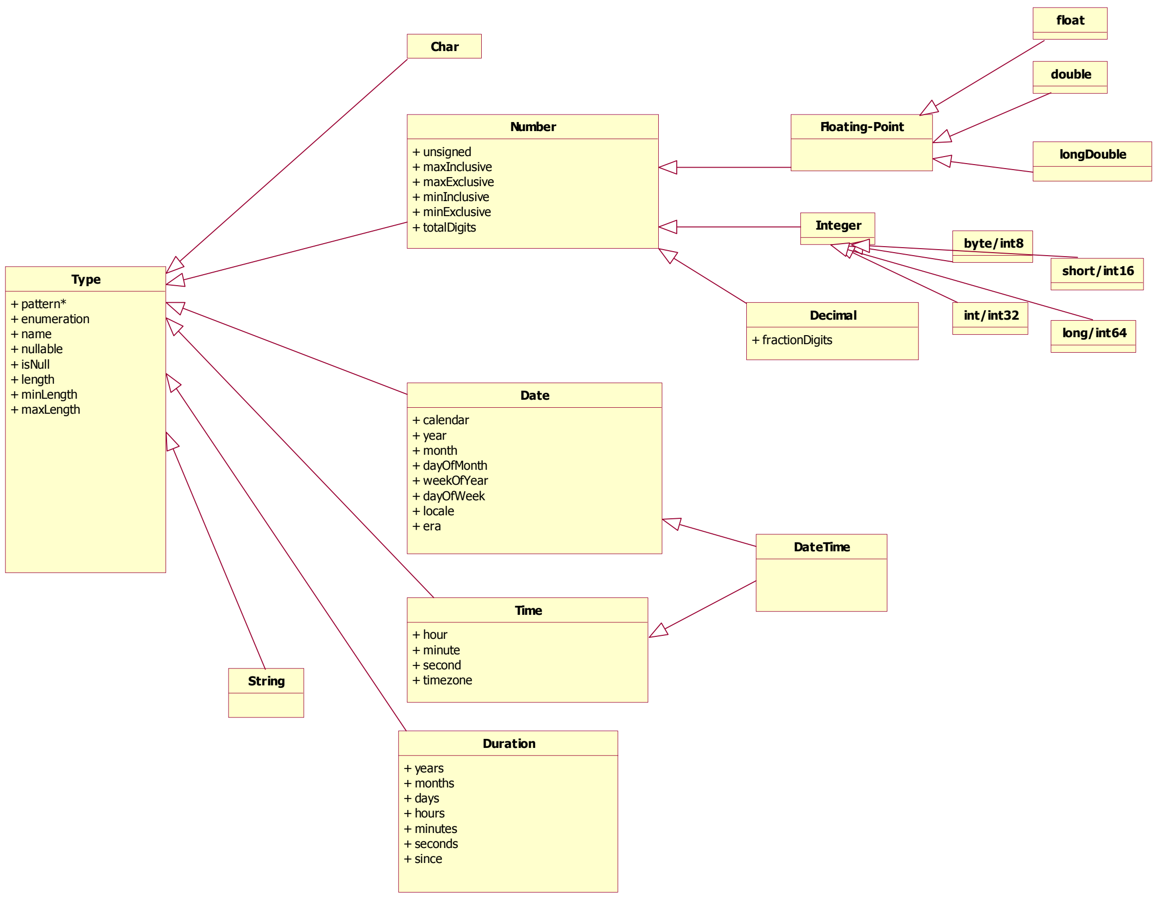
5. DFDL Information Model

This idea comes from the XML Information Set. The XML information set is a way of separating concerns about the structure of XML data from the semantics of that data. For DFDL we do much the same by describing a DFDL information model which captures the kinds of data that can be expressed in DFDL but is completely independent of the representation of that data.

DFDL describes data assuming certain information content in that data. We will use the term DFDL Information Set to refer to the set of items within this model.

5.1 Built-In Types

The following diagram is a UML class diagram showing the basic or primitive types in the DFDL Information Set. Some of the types have attributes shown and these attributes are those that are about the information content, and not about the representation of these types.



In the above notice that there is no notion of character sets. The ‘Char’ datatype is an abstract character. Think of it as a Unicode character code, which is really an integer.

The attributes of the ‘Type’ class deserve some comment. A type can be named, which is not particularly controvercial. A type can have an attribute which gives an enumeration of legal values for that type. This is a semantic validation constraint. In addition a type can have a pattern. This is a semantic constraint only for types where it is the nature of the type for a pattern to be meaningful. Of the basic/primitive types this is only sensible for the type ‘String’, so it seems as though the pattern attribute should be part of the ‘String’ class; however, this same attribute will become important for other types once we introduce the representation layer; hence, it appears on the parent class ‘Type’.

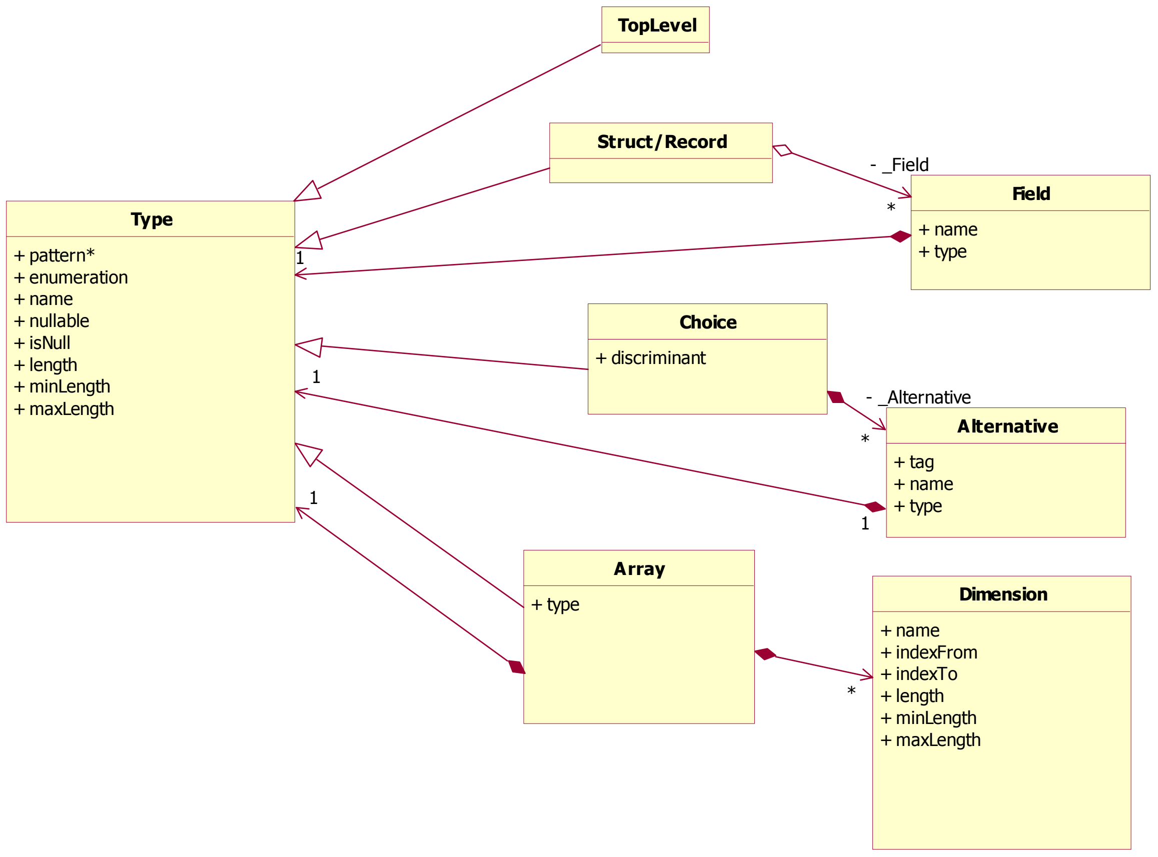
The notation 'pattern*', i.e., the star after the attribute name, indicates that this attribute can be specified multiple times. For example, in the case of pattern, if several patterns are given then valid data can match any one or more of the regular expressions.

(TBD: revise this "*" to use standard UML <<0..*>> notation.)

The String type is special. It isn't really a primitive, but rather is a standard interface to many kinds of one-dimensional arrays of characters.

5.2 Type Composition

In addition to these basic/primitive types, there is a type composition mechanism that allows more complex information items to be described. These mechanisms are shown here:



The above shows the kinds of fundamental type constructions available in the information model. These are struct/record, which is a collection of named fields each potentially of a different type, arrays which can have multiple dimensions, but all elements are of uniform type, and choice structures. Choice structures are called unions in some programming language terminology. They're also known as variant structures or variant records. A choice type represents one of a number of possible alternatives, any of which could be contained in a given type, but in any given instance of an information item only one alternative can actually exist.

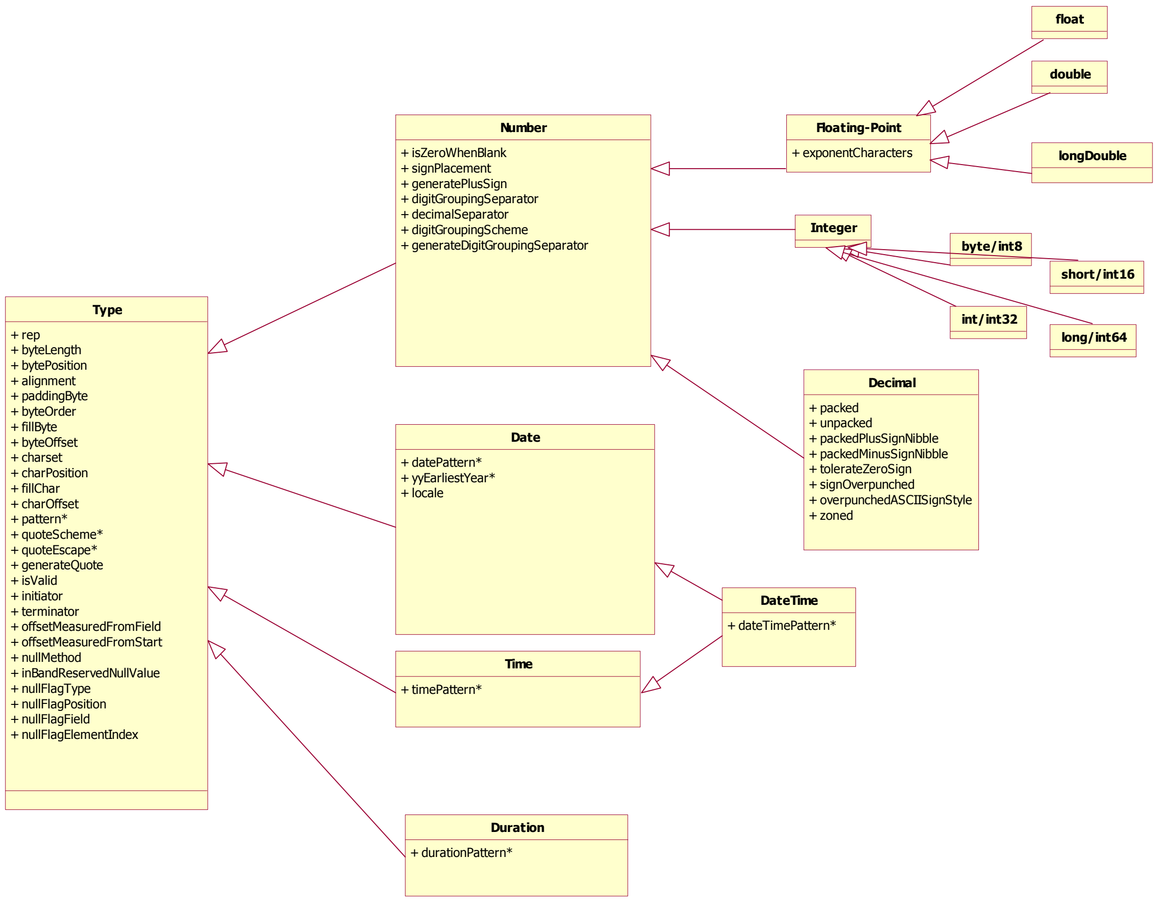
DFDL captures the structure of data, and the types of the elements it is made up of. It does not capture relationships between items in the data outside of a few critical relationships:

- Hierarchical containment
- Variant: that is, one or a few fields store the information by which one is able to discriminate a choice/variant.

Another way to look at this is that DFDL omits constructs analogous to the Xlink and Xpointer capabilities in XML. Those mechanisms allow one to capture arbitrary non-containment relationships between objects. An arbitrary graph of objects can be serialized into XML in such a way that it's topology is represented. This is more expressive power than is needed in DFDL, at least for commercial applications.

5.3 Representation Layer Attributes: Basic Types

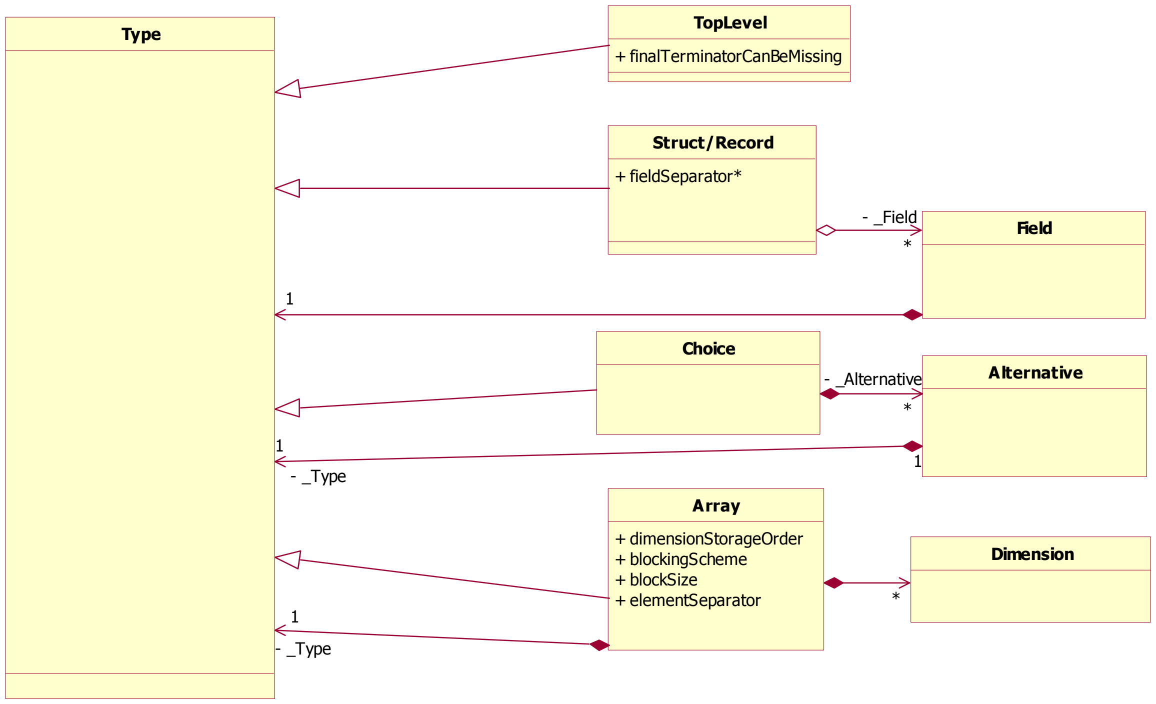
The basic/primitive types have a number of attributes which have only to do with the representation of these types. These are shown on the diagram below:



These attributes are discussed in detail in later sections of theis document.

5.4 Representation Layer Attributes: Composed Types

In the more complex types there are also additional attributes used to express representation. These are shown on the diagram below, which omits all the attributes already discussed for the pure information set.



Attributes marked with '*" are those where the DFDL description can have multiple defintions. E.g., a Struct can have multiple field. Each field separtor is an alternative. All of them are considered equivalent when describing existing data being read. The first field separator specified is considered the primary one and it is this primary one that is used when an application is writing out data.

Later sections of this document will describe each of these types and their attributes in detail.

5.5 Typical Commercial Data Characteristics

(TBD: move this section elsewhere)

The central concept in data descriptions in commercial applications is the concept of a 'record', and a file containing records. That is, the vast majority of files in commercial data processing applications hold at top level a single long 1-dimensional array of records where the length of the array is determined only by the end of the file.

Less frequently, but still important are files with header and trailer records that differ from those found in between. Such a file looks at toplevel like a single record having a header field, itself a record with substructure, a body field which is an array of records, and a trailer field which is itself a record.

6. Example 1

Rather than delve into the details, let's try for intuition-building by example. This example is a DFDL descriptor and some data records all in one XML file. The data format corresponds to a Cobol FD borrowed from the OMG CWM Specification.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE dfdl:dfdl SYSTEM "http://www.ggf.org/2003/DFDL-std-entities.dtd">
<dfdl:dataset xmlns:dfdl="http://www.ggf.org/2003/DFDLSchema"
              xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
              xsi:schemaLocation='http://www.ggf.org/2003/DFDLSchema'
              xmlns:cust="http://www.bigcompany.com/custinfo"
         xmlns:doc="http://www.bigcompany.com/datadoc">

<!-- This all works just like XSDL. I.e., the way annotations work,
namespaces, etc. -->
```

```
<dfdl:annotation>
<dfdl:documentation>
<doc:origin system="ibm-390" planguage="cobol" compiler="VSCobol"
version="14">
01 Customer.
    05 account PIC 999999
       USAGE BINARY.
    05 custName PIC X(50).
    05 custAddress.
        10 address1 PIC X(80).
        10 address2 PIC X(80).
        10 city PIC X(30).
        10 state PIC X(3).
        10 postcode PIC X(11).
        10 country PIC X(20).
        10 filler PIC X(2).
</doc:origin>
</dfdl:documentation>
</dfdl:annotation>

<dfdl:defType name="Address">
   <dfdl:defaults type="string"/>
   <dfdl:struct>
      <dfdl:field name="address1" length="80"/>
      <dfdl:field name="address2" length="80"/>
      <dfdl:field name="city" length="30"/>
      <dfdl:field name="state" length="3"/>
      <dfdl:field name="postcode" length="11"/>
      <dfdl:field name="country" length="20"/>
      <dfdl:field name="filler" length="2"/>
   </dfdl:struct>
</dfdl:defType>

<dfdl:defType name="Customer">
   <dfdl:struct>
      <dfdl:field name="account">
            <dfdl:type name="int32" rep="binary" byteOrder="bigEndian"/>
      </dfdl:field>
      <dfdl:field name="custName" type="string" length="50"/>
      <dfdl:field name="custAddress" type="Address"/>
   </dfdl:struct>
</dfdl:complexType>

<dfdl:toplevel name="custInfoFile">
  <dfdl:defaults charset="ebcdic-cp-us"/>
  <dfdl:array type="Customer">
    <dfdl:dimension/>
  </dfdl:array>
</dfdl:toplevel>

<dfdl:data whereLocated="here" encoding="hexBinary">
02f572a320202020202020202020202020202020202020202020202020202020202020204173
63656e7469616c20536f667477617265202020202020202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
35302057617368696e67746f6e2053747265657420202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
202020202057657374626f726f756768204d41202020202020303137353120202020202020202020
2020202020202020205553410d0a02f572a320202020202020202020202020202020202020202020
202020202020202020202020417363656e7469616c20536f66747761726520202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
202020202020202020202020202035302057617368696e67746f6e205374726565742020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
20202020202020202020202020202020202020202020202020202020202020202020202020202020
2020202020202020202020202020202020202057657374626f726f756768204d41202020202020
2030313735312020202020202020202020202020202020205553410d0a
</dfdl:data>

</dfdl:dataset>
```

The important things to look for in this example are:

1. note that the annotation, including the Cobol FD text as documentation, works exactly like XSDL. This is a general principle. XML documents need the ability to attach various metadata to XML Schemas. DFDL is no different, so we just clone the XSDL mechanisms.
2. we've included the data immediately in the file. Other options would be used for separate data.
3. We set a default for the 'type' attribute at line NNN, and this applies throughout the scope unless redefined at another closer level. Specific fields pic up the default and further refine the type.

7. Example 2

This example is a translation of an example from a paper about a proposed data format description language called BinX. (see http://www.epcc.ed.ac.uk/~gridserve/WP5/Binx/sci-data-with-binx.pdf for the original version of this example.)

This example is much more typical of scientific computing. The data format being described is a 2d array of complex numbers.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dfdl:dfdl SYSTEM "http://www.ggf.org/2003/DFDL-entities.dtd">
<dfdl:dataset xmlns:dfdl="http://www.ggf.org/2003/DFDLSchema"
              xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
              xsi:schemaLocation='http://www.ggf.org/2003/DFDLSchema'
              xmlns:cust="http://www.bigcompany.com/custinfo"
         xmlns:doc="http://www.bigcompany.com/datadoc">
     byteOrder="bigEndian">

<dfdl:deftype name="complexNumber">
  <dfdl:struct>
    <dfdl:field type="float" name="real"/>
    <dfdl:field type="float" name="imaginary"/>
  </dfdl:struct>
</dfdl:complexType>

<dfdl:toplevel>
  <dfdl:struct>
    <dfdl:field type="float" name="inputParameter1"/>
    <dfdl:field type="int32" name="inputParameter2"/>
    <dfdl:field>
      <dfdl:array blockingScheme="FB" blockSize="32K"
                  type="complexNumber" dimensionStorageOrder="rowMajor">
        <dfdl:dimension indexFrom="0" indexTo="99" name="x"/>
        <dfdl:dimention indexFrom="0" indexTo="4" name="y"/>
      </dfdl:array>
    </dfdl:field>
  </dfdl:struct>
</dfdl:toplevel>

<dfdl:data whereLocated="files">
  <dfdl:dataFile url="http://www.epcc.ed.ac.uk/testFile.bin"/>
</dfdl:data>
</dfdl:dataset>
```

Some notes about this example:

- Notice the use of anonymous types and fields.
- Noice the use of blocked data, i.e., the blocksize and blockingScheme attributes.

8. Dynamic Expression Language

The values in our information set are computed from the representations of the information items in the actual data representation. Sometimes the way that the values are computed is obvious and clear. E.g., for a 4-byte long binary big-endian integer, the way the bytes are interpreted to compute the value is assumed to be built-in to the DFDL processor.

In other cases it is necessary for the DFDL descriptor of data to contain expressions, and possibly even programatic logic to show how to extract certain information from the representation so as to compute the value of an information item or an attribute of it. For this reason DFDL includes a expression language.

Our expression language is a small pure-functional language. Its expressions allow:

- use of field values by using the field name in the expression
- arithmetic: +, -, *, /, max, min, abs, <<, >>, ... on all numeric data types.
- bit logic: AND, OR, NOT (twos compliment), XOR on all integer data types having binary representations
- boolean logic
- conditional expressions: if-then-else
- local variables, a variety of iteration constructs.
- (TBD: any sort of lists/tuples or multiple args/returns?)

This language allows a declarative expression of the functions we need to make sense of the data. This goes well beyond just allowing us to express the lengths of variable length arrays as we'll see in the next example.

(TBD: a section on the built-in functions in this language on our built-in types.)

The DFDL expression language is inspired by the design of the Xquery language. Examples later in this document will show the expression language at work in handling the interpretation of complex data types.

(TBD: Implications for input/output symmetry? I.e., how to invert the information in these expressions so that one can write data in the described format as well as read it? It's certainly not feasible to invert in general, since the expression language is turing complete a one-way function could be written; however, we either need to know we can invert most cases, or we need a set of idioms for describing how to write as well as how to read the data.)

9. Attributes

9.1 Attributes of the 'Type' Class

9.1.1 DFDL Information Set Attributes for the 'Type' Class

These attributes are independent of representation issues.

- pattern (*): A regular expression that a text representation must match to be considered valid.
- enumeration: a set of values that valid data of this type make take on to be considered valid.
- name: optional. A name for this type.
- Nullable: determines whether the type allows for null values.
- IsNull: this attribute is used at run time to inquire as to whether the value of a specific data item is in fact null or not.
- Length - One might think this attribute would apply only to arrays and strings; however, since most (all?) types can have text representations, these attributes are factored up to the base class. That is, from the pure information-set perspective, the length attribute belongs on the array dimension class; however, due to representation issues it gets located here on the type base class instead.
- MinLength
- MaxLength

9.1.2 Representation Attributes for the 'Type' Class

These attributes are about representation issues.

- rep (representation): Can be text (the default) or binary. Text indicates that the data is in textual format (and implies a charset attribute).

9.1.3 Binary Representation Attributes

- ByteLength: It must always be possible to compute the length of any data item. There are a variety of means by which the byteLength may be calculated. In general this scheme is called the "length method". The schemes are:
    - Fixed: a literal constant specifies the length, or the length is known implicitly based on the type (e.g., binary int32 takes 4 bytes)
    - Delimited: initiator, terminator, fieldSeparator, elementSeparator are used to determine the start and end of a data item. This combined with character set information allows the byte length of text fields to be calculated.
    - Stored: a field or fields contains the byteLength, or data from which the byteLength can be calculated.
    - Implicit: the length of the item is the sum of the lengths of all recursively contained items. That is, sometimes aggregate items have their own stored length for the overall composition of sub items, and sometimes aggregate items have no such length information.
    - *(TBD: do fixed or stored lengths include or exclude the size of the padding added on to meet an alignment requirement?)*
- bytePosition: This is specified as an absolute byte offset from the beginning of the record (assumed to be the same as the beginning of the first named field of the record)
- byteOffset: This is an offset from the beginning or end of any prior field of the record. The default value is zero bytes from the end of the previous field.
- alignment: This is an integer greater than zero. It gives the alignment required for the 1st byte of the item. The default value is equal to the largest byteAlignment specified on any item specified recursively within the field, or 1 if none is specified. The value is required to be at least as large as this default. If larger it must be a multiple of this default. Conforming DFDL implementations must detect inconsistent alignment specifications.
- paddingByte: This attribute gives a byte integer value used to fill in parts of a field that are not filled in by the data type. This occurs when you have variable length data types with a maximum length represented as a fixed size buffer of the maximum possible size and a stored length field or delimiter indicating the amount of the buffer actually being consumed by this data item. (Happens all the time in Cobol data.) In this situation the paddingByte value is used to fill in the unused bytes of the representation. This is important for security reasons.
- fillByte: The positions of the fields in a record and the sizes of those fields may leave some bytes of the record unpopulated with data; that is, they are unused. When writing such records out this attribute specifies that they are to be filled with a specific byte value. It is crucial to secure software that they not be left to contain whatever was in memory. The default value of recordFill is 0 (zero). This byte value is also used to fill any unused bytes due to alignment. Non-zero values (e.g., using the character code for 'X') are mostly used as a debugging technique so that unused space or mistakes in the DFDL descriptor are glaringly obvious when examined with simple tools like print statements.

  (Note: that if DFDL supports individual bit fields then it must not be possible to write some bits of a byte without specifying that all other bits of that byte have specific initial values (usually all zero.))

- byteOrder: bigEndian or littleEndian. Valid and applies to binary multi-byte representations and text representations when the charset is a fixed-width, multi-byte encoding, such as UTF-16 or UTF-32 which has big and little endian variants.

  (TBD: there's some mixed endian format out there too. Vax-endian?. Do we need it.)

  (TBD: UTF-16 can have a byte-order-mark or BOM as the first 2 bytes. This allows a run-time dynamic mix of big and little endian UTF-16. Do we need/want this? This would be per-field BOMs.)

9.1.3.1 Text Representation Attributes

- charset: for text format, indicates the character set name. This is an IANA character set name. (TBD: or the name of a character set map from some other standard?) Note that in XML the entire XML file has a character set specified by XML conventions. In DFDL we must deal with field by field specification of character sets in order to support applications that mix binary and character data and that convert that data from one character set to another.
- charPosition: This is meaningful only if the entire record contains nothing but text fields. This is the offset from the 1st character in the record. (The first character is in characterPosition=0.)
- charOffset: This is an offset from the beginning or end of any prior field of the record. The default value is zero.
- FillChar: Similar, but for character strings.
- quoteScheme(*): Specifies the opening and closing quotation marks or brackets that can surround the value of an item in text representation. If a quoteScheme is specified, it contains two characters. The first is the opening quote mark, the second the closing. If only one character is specified it is as if both characters were specified as the same value. That is, the same character opens and closes. Inside the quote marks the delimiters (fieldSeparatorter, terminator, initiator, elementSeparator) may be found unescaped.
- quoteEscape(*): Specifies the prefix character used to escape occurances of the quotation characters if they appear inside the value of an item.
- GenerateQuote: When writing data out this attribute specifies that the quotation marks (or brackets) are to be placed surrounding the value text for an item.
- Initiator - a character or string used to delimit the start of a data item.

9.1.3.2 Both Binary and Text Representation Attributes

- offsetMeasuredFromField: specifies the field that the offset (characterOffset or byteOffset) applies to. Default is to the preceeding field in the DFDL description.
- offsetMeasuredFromStart: boolean. Determines whether the offset is from the start or end of the specified field. Default is false meaning end of field.

It is possible to interpret the same storage multiple ways by specifying multiple fields having the same bytes within the data.

- Terminator - Usually this is a character or string used to delimit the end of a data item. However it can also be a binary value used to delimit the end of an array. For example, an array of integers terminated by two final integers both zero. (Unix process environment data structures are terminated this way. So if you wanted to describe a Unix core dump in DFDL this is one way you could handle the environment part.)

9.1.3.3 Nullability Attributes

- NullMethod: values are either inBand or flag. If inBand is specified, then the attribute inBandReservedNullValue must be defined. If flag is specified then the attributes nullFlagType, nullFlagPosition, nullFlagField, and nullFlagElementIndex are used.
- InBandReservedNullValue: this attribute specifies a reserved value that is used to indicate null. InBand null representations often must reserve one or more values to be null indicators.
- NullFlagType: the type of the representation of the null flag.
- NullFlagPosition: the byte position of the null field. (TBD: absolute/relative, w.r.t. this field or the beginning of the record?)
- NullFlagField: the name of the field containing the null flag.
- NullFlagElementIndex: if the nullFlagField is an array (one dimensional) then this attribute gives the index into that array where the null flag will be found.

9.2 TopLevel Type

- finalTerminatorCanBeMissing: Mostly used for line-oriented records where the last line of a file might be missing the CRLF or LF. Avoids an error or diagnostic message when this is acceptable. Default to true if the delimiter is LF or CRLF. Otherwise the default is false.

9.3 About Delimiters: Initiators, Separators, and Terminators

In textual representations data items can be delimited by characters or strings of characters. There are three different kinds of delimiters: initiators, terminators, and separators. Individual types can have initiators and terminators. Only aggregate types can have separators. (Aggregate types = arrays and struct/records).

A terminator marks the end of an item. For example, a variable-length string as represented in a C program is a vector of characters terminated by the NUL character code (NUL is zero). A file may contain a vector of records where each record is a non-blank line. The vector is terminated by a blank line.

A separator is positioned between two items to mark their boundary. Comma, tab, and pipe separated values are popular file formats. Each line in such a file contains a single record whose fields are separated by the specified character.

An initiator delimits the beginning of an item. For example a textual tag such as "[coordinates]" might preceed the data lines containing coordinate data.

For any delimiters, the value specified is either a single character or a string (usually short) specified in the charset of the field. That is, if the data being terminated or separated is in say, the EBCDIC-CP-US charset, then the string specified for a delimiter is either

- Specified as a string in the character set of the XML DFDL document
- Specified as a vector of bytes containing the character codes in the EBCDIC-CP-US character set.

If a regular string is specified, then the characters are translated from the character set of the XML DFDL document into the target character set. For example, a fieldSeparator=”,” specification on EBCDIC-CP-US fields results in the single byte 0x2C regardless of the fact that the Unicode UTF-8 code for the “,” is 0x6B.

In case of non-printing characters XML escape sequences can be used to specify the numeric value of the character codes. See also the appendix "About Literal Strings".

(TBD: details of how you specify bytes, not characters?)

(TBD: why not allow regular expressions for all these?)

9.4 Struct/Record Type and Field

The most common aggregator for multiple items is the 'struct' or ‘record’ construction where each data item is called a field. Note that since each field has position information for its representation the order of fields in the struct definition does not necessarily mean the representations of those fields are in the same order within the record data. By default they will be, but they can also be scattered all over the representation.

An example of a struct is the assembly of multiple fields to form a subrecord. For example, a name, street address, city, state, and zipcode field can be structd to create a subrecord type that is named 'USAddress'

- FieldSeparator(*) This is a character or string found between fields of the struct/record.
- recordTerminator(*) This is a character or string found after the last field of the struct/record.

Use of the recordTerminator or fieldSeparator attributes implies that all fields within the record have the text representation attribute. Most commonly the recordSeparator is an ASCII LF or CRLF sequence and is used for data that is represented textually with each textual line being a record. However, any character or string could be used.

If a fieldSeparator and recordTerminator are the same, or the recordTerminator is contained within the fieldSeparator, or the recordTerminator appears inside the text of a field, then you can detect this and then processing the record to find its end requires parsing the field in question and then scanning for the recordTerminator. Note that the last field of the record can’t contain the recordTerminator in its text since it is not followed by a fieldSeparator.

There are cases where the recordTerminator and/or fieldSeparators are not literal, but are stored in fields of a header-record within a data file. Hence, it is sometimes necessary to compute the value of these attributes using an expression that extracts the values of other fields.

9.5 Choice Type and Alternative

- Discriminant: The value of this attribute is an expression that computes the value of the tag expression of one of the alternatives. Commonly this expression is simply the name of a prior field in the record the value of which is used to determine the alternative.
- Alternative: tag - This expression's value is the value the discriminant expression must have in order to select this alternative. Most commonly this is a literal constant.
- Alternative: name - This is optional and not used by DFDL. It provides documentation.
- Alternative: type: The type of the data contained in this alternative.

9.6 Array Type and Dimension

- DimensionStorageOrder: rowMajor or columnMajor.
- Type: the type of the elements of the array.
- blockingScheme: This describes an organization of the data items of arrays into blocks. (A file containing a top level sequence of records is a one-dimensional array.) The schemes are named U, F, FB, V , VB , VBS. Format ‘U” means non-blocked, in which case the records are simply one following the next. This *attribute* is optional and defaults to non-blocked. This *attribute* is used to support legacy data blocking formats still in use today. See the appendix: "Record Blocking Schemes Detailed" for a definition of the values for this *attribute*.

  Note that it is possible for one data file to contain data with several kinds of blocking. This can occur if two files of binary data with different blocking schemes are simply concatenated to form a larger data file.

  (TBD: all ‘V’ data has an implicit 4-byte binary (? Check that it’s not decimal) length preceeding the variable-length record’s data. How is this implicit field accessed?)

- BlockSize: . See the appendix: "Record Blocking Schemes Detailed" for a definition of this *attribute*. This is equivalent to the IBM BLKSIZE used in JCL.
- (TBD: might need an equivalent to LRECL)
- ElementSeparator(*) - In text representation, this character or string is used to separate elements of the array.

9.6.1 Dimension

- Name: This is for documentation purposes.
- IndexFrom: Defaults to zero. This allows the indexing of a dimension to be across a range of continuous integers that does not begin at zero. E.g., you can index a dimension containing 11 items using indices from -5 to +5
- IndexTo: The upper bound (inclusive) on the index of the array. To clarify, the number of elements in the dimension is indexTo - indexFrom + 1.

9.7 Number Type

- Unsigned
- MaxInclusive
- MaxExclusive
- MinInclusive
- MinExclusive
- totalDigits
- isZeroWhenBlank: A common data convention for text representation is that if all characters are blank then the number should be assumed to be zero even though technically it's not valid decimal unless it has at least the single digit ‘0’.(OMG CWM Vol2, Section 3 Cobol)
- signPlacement: 'leading' , 'trailing', or ‘surround’ signs are supported. Default is leading for text, but trailing for decimal binary forms. Surrounding sign means that “(5)” is equavalent to –5. Surrounding sign is only for text representations.
- generatePlusSign: For text representations create a '+' sign in the output (It's always optional on input.)
- (TBD: does XML allow hex strings for integers?)
- digitGroupingSeparator: like POSIX thousands_sep from Locale settings. Default is depending on the Locale settings.
- DecimalSeparator: like POSIX decimal_point from Locale settings. Default is depending on the Locale settings.
- DigitGroupingScheme: like POSIX 'grouping' from Locale settings. Default is depending on the Locale settings.
- GenerateDigitGroupingSeparator: when writing data, this boolean indicates whether digits are grouped or not. Default is false.

Example: In the EN_US locale one might encounter 101,234,567.89. In a FR locale, the equivalent format would be 101.234.567,89, and in a Thai locale, 10,12,34,567.89. Notice that in the Thai locale, the "thousands separators", a.k.a., digitGroupingSeparators don't get used to separate thousands; hence, we don't copy the misleading POSIX name.

(TBD: official name of EN_US, FR_??, and Thai locales?)

9.8 Floating Point Type Properties

- exponentCharacter: Allow 'F' and 'G' in addition to 'E' for the exponent marker. (Upper and lower case allowed, i.e., 'EFGefg' are the characters allowed.)

9.9 Decimal Type Properties (Summary)

A large area of difference between commercial and scientific data comes in the kinds of numbers being processed. Commercial applications make heavy use of decimal arithmetic where the precision is larger than what can be captured with today's floating-point representations. Usually this data is fixed precision. That is, the total number of digits and number of fractional digits are fixed. This is sometimes called 'fixed-point' arithmetic; however, it is also critical that the arithmetic itself be done in base 10 with base 10 rounding. There are a variety of commonly used representations for decimal numbers including both string-based and binary (the packed and unpacked decimal) forms. There is also a relatively new standard for decimal arithmetic and representation, though to date this is hardly used anywhere. The decimal standard is IEEE 854. It is being merged with IEEE 754, the standard for binary floating-point arithmetic. (See http://grouper.ieee.org/groups/754 as a starting point for details.)

The need for integration of large numbers of software systems into grids puts a premium on the ability to comprehensively handle legacy data formats and to freely intermix them with more modern representations. The ability to generate these formats as well as to consume them is an important requirement.

There are lots of nasty details associated with legacy decimal formats. These details are only important to those interested in these legacy formats hence, they've been put in an Appendix: Decimal Number Attributes Detail. However, here is the list of the attributes needed for specification of decimal formats.

- fractionDigits
- packed
- packedPlusSignNibble
- packedMinusSignNibble
- unpacked
- zoned
- tolerateZeroSign
- signOverpunched
- overpunchedASCIISignStyle

These are used along with the signPlacement (leading or trailing), byteOrder, text/binary representation, and charset specifications to provide full control over the representation of decimal numbers. Details on these attributes are in the appendix.

(TBD: For the future, due to IEEE 854 do we need a decimal of arbitrary size with a floating point?)

9.10 Date and Time Properties (Summary)

Handling date and time is a thankless job. There is an enormous variety of data formats in wide use for representing dates and times. Standards like ISO 8601 standard date and time formats have been very late entrants to the world of data processing.

The proper capture of date and time information in DFDL involves not only describing the information stored in the data, but also describing much contextual information that is needed to understand the data but is not stored. For example, if the data is in a file and it contains dates and times in the format yyyy/mm/dd, hh:mm:ss, then we must know what time zone is assumed and also what calendar system is implied for the date. It is likely that data in this format is in the Gregorian calendar, which is by far the most important calendar for commercial purposes, but the time zone is completely ambiguous. (Other calendars may be more important for scientific applications where data sets may be about times far in the past or far in the future.)

9.10.1 Date and Time Attributes

These are largely derived from the ICU software library (International Components for Unicode).

A date, time, or combined datetime field contains a subset of the information needed to properly interpret the data. The complete set of attributes of a date and time is given below.

This set seems large, but ultimately it must provide all the information to unambiguously initialize a date or time object in a modern programming language. For example Java allows specification of the calendar and the initiation date of the calendar.

See also the Appendix: Ambiguity of Date/Time Information

- *Calendar: Gregorian, with start date 15 Oct 1582 by default.*
- *Era*
- *Year*
- TBD: Quarter
- *Month*
- *DayOfMonth*
- *WeekOfYear*
- *DayOfWeek*
- *DatePattern(*)*- See the Appendix: Date, Time, and Duration Patterns for details of this syntax.
- *yyEarliestYear(*)* - When converting two digit year representations into actual year values this attribute is used. It's value is the earliest year to be represented. E.g., if yyEarliestYear=1970, then the two digits "76" refer to the year 1976, but the digits "66" refer to 2066.
- *Locale*
- *Hour*
- *Minute*
- *Second*
- *Timezone*
- *TimePattern(*)*- See the Appendix: Date, Time, and Duration Patterns for details of this syntax.
- *DateTimePattern(*)* - See the Appendix: Date, Time, and Duration Patterns for details of this syntax.

9.11 Duration

- since: specifies that the duration is the length of a time interval which started at a specific date/time. The value of this attribute must be given as an ISO date/time. The duration interval is only clearly unambiguous for days since 1926-01-01T00:00:00Z. Durations since dates prior to that are ambiguous depending on the locale involved and when the country converted to the Gregorian calendar.

- Years
- Months
- Days
- Hours
- Minutes
- Seconds
- DurationPattern(*) - See the Appendix: Date, Time, and Duration Patterns for details of this syntax.

10. Defined Types

We follow the convention herein that user-defined type names start with an upper-case letter and field names, attribute names, enumerated values, and any other attribute names start with a lower-case letter.

Named types are defined by use of a "deftype" construct. The type constructors introduced above (array, struct/record, nullable, and choice) are used.

In most places where a type can be expressed in DFDL one can use the name of a defined type, or one can directly include an anonymous type definition.

(TBD: also need ability to create a named "group" which is a fragment of a type. E.g., that way you can share defininitions of common parts of record formats without having to define types for those parts. E.g., suppose you have 3 record types each of which have the same two key fields. The two key fields could be defined once as a named group and included in the definitions of the 3 record types. The two key fields don't make up a named type themselves.)

10.1 Example: Stored Length of Repeating Subrecord Array

Continuing our practice of examples before formalism, lets examine a complex type definition where the length of a variable-length array is stored in a field. The Cobol "OCCURS DEPENDING ON ..." construct creates exactly this kind of data.

Consider this Cobol FD fragment:

```
01  PATIENT-TREATMENTS.
       05  PATIENT-NAME                PIC X(30).
       05  PATIENT-SS-NUMBER           PIC 9(9).
       05  NUMBER-OF-TREATMENTS        PIC 99 COMP-3.
       05  TREATMENT-HISTORY OCCURS 0 TO 50 TIMES
              DEPENDING ON NUMBER-OF-TREATMENTS
              INDEXED BY TREATMENT-POINTER.
           10  TREATMENT-DATE.
               15  TREATMENT-DAY        PIC 99.
               15  TREATMENT-MONTH      PIC 99.
               15  TREATMENT-YEAR       PIC 9(4).
           10  TREATING-PHYSICIAN       PIC X(30).
           10  TREATMENT-CODE           PIC 99.
```

(Source http://www.discinterchange.com/reading_COBOL_layouts_.html)

In this data the subrecord named ‘TREATMENT-DATE’ appears between 0 and 50 times depending on the value of the prior field ‘NUMBER-OF-TREATMENTS’, which in this record is type ‘COMP-3’ which is packed decimal representation.

DFDL expresses this same record layout as follows:

```
<deftype name='PATIENT-TREATMENTS'>
  <defaults rep='text' unsigned='true'/>
  <record>
```

```
     <field name='PATIENT-NAME'><dfdl:type name='string' length='30'/></field>
     <field name='PATIENT-SS-NUMBER'>
             <dfdl:type name='decimal' totalDigits='9'/></field>
     <field name='NUMBER-OF-TREATMENTS'>
              <dfdl:type name='decimal' packed='true' totalDigits='2'/></field>
     <array name='TREATMENT-HISTORY'>
           <dimension minLength='0' maxLength='50'
                      length='{ NUMBER-OF-TREATMENTS }'/>
           <record>
             <field name='TREATMENT-DATE'>
                <record>
                  <field name='TREATMENT-DAY'>
                         <dfdl:type name='decimal' totalDigits='2'/></field>
                  <field name='TREATMENT-MONTH'>
                         <dfdl:type name='decimal' totalDigits='2'/></field>
                  <field name='TREATMENT-YEAR'>
                         <dfdl:type name='decimal' totalDigits='4'/></field>
                </record>
             <field name='TREATING-PHYSICIAN'>
                         <dfdl:type name='string' length='30'/></field>
             <field name='TREATMENT-CODE'>
                         <dfdl:type name='decimal' totalDigits='2'/></field>
           </record>
      </array>
      </record>
   </deftype>
```

In this DFDL notice that we're able to express what the Cobol FD expresses, albeit in many more characters because DFDL is an XML-based language. Unlike the Cobol FD; however, in DFDL we have places to put the additional information that the Cobol leaves unstated, for example, what byteOrder is used to store the packed decimal number 'NUMBER-OF-TREATMENTS'.

The field 'NUMBER-OF-TREATMENTS' is used to determine the length of the array of subrecords via the expression '{ NUMBER-OF-TREATMENTS }'. The braces are what introduce expression evaluation. (This style comes from Xquery.) This implies that a string literal containing braces must use escape sequences. The value of a field is produced by simply using the name of the field in the expression.

In our example above the expression is nothing more than a field name, but they can be substantially more complex. The only rule is that they cannot refer to the names or attributes of items whose values cannot be determined without knowing the value of the attribute that is being defined. That is, the length of a field cannot be stored in a field that appears after it in the record since its position would then be ambiguous.

10.1.1 Example: Variable-length Strings

The following DFDL defines a variable length string type where the length of the string is stored in a 4-byte integer which preceeds the variable-length array of characters.

```
   <deftype name='VLString'>
     <record>
       <field name="len">
         <dfdl:type name='int32' rep='binary'
                                 byteOrder='bigEndian'/></field>
       <array name="content" type='char'>
              <dimension length='{ len }'>
       </array>
     </record>
   </deftype>
```

In this example, we use an array of characters. Note however that some character sets contain fixed-width characters and some character sets contain variable-width characters. In the worst case here, say, if the charset were specified to be UTF-8, then this would be a variable-length array of characters each of which is variable width.

(TBD: should there be native stored-length capability in the built-in string type?)

10.2 Choice and Alternatives

Discriminated variants are the term we used for what the programming language community calls tagged and untagged union types. The concept is that you need to examine some data field or fields to discriminate whether certain fields of the record are present or absent. Discriminated variants come into existence due to constructs like the Cobol REDEFINES clause.

Consider this example. Suppose we have a very clever variable-length string representation that does not use delimiters. The representation for strings whose length is 128 bytes or less is a single byte prefix length field, followed by a variable-length array of characters. For strings longer than 128 bytes, the representation contains a big endian int32 followed the variable-length array of characters. The smart string type is one or the other of these two representations.

Unfortunately, these two representations are ambiguous as so far described. There is no reliable way to determine which of the two representations is in use. That is, there is no sure fire *discriminator* possible. To fix this we add the additional requirement that when the length is 128 or less we flip the most significant bit of the prefix byte, and specify that the maximum length for the larger representation is 2^30-1 bytes.

Now let’s look at the DFDL to describe this data type:

```
<deftype name='SmartString'>
  <record>
     <choice discriminator='{if ((byteLen and 0x80) = 0) then 2 else 1}'>
         <alternative tag='1'>
             <field name='byteLen' type='byte'/>
         </alternative>
         <alternative tag='2'>
             <field name='int32Len' type='int32' alignment=’4’/>
         </alternative>
     </choice>
     <array name='content' type='char'>
          <dimension length='{if ((byteLen and 0x80) = 0)
                                  then int32Len else (byteLen and 0x7F)}'/>
     </array>
  </record>
</deftype>
```

In this definition, we see a choice between two different representations of the stored length. The discriminator expression tests the most significant bit of the byteLen field, which is also the most significant, and unused, bit of the int32Len field. This bit is the discrimination flag for our two variants.

We also see that the length attribute for the variable-length array is computed by an expression. The length isn’t directly stored in a field in this case, we must decode it from what is stored.

Note that in Cobol, the REDEFINES construct generally automatically pads all the variants to the same length. (This was made the default behavior in 1985 for Cobol, before that time most applications padded all redefines carefully to the same length to insure that their overall record was of fixed length.)

In DFDL, if the alternatives are all to be padded to the same length then the appropriate attributes must be set to indicate the padding. (For example, using byteLength and fillByte explicitly.) As expressed in the above example fields of type ‘SmartString’ will be variable length not only because of the array of characters, but because the stored length field itself is of varying representation. A SmartString of length 127 characters in a single-byte-wide charset takes up a total byteLength of 128 bytes. However, a SmartString of length 128 takes up at least 132 bytes because of the increased size of the prefix stored length field. The actual number of bytes consumed by this SmartString may be more than 132 because of the alignment to a 4-byte boundary required in the type definition for the 4-byte prefix stored length field.

Often variants are discriminated based on an explicit field, called a tag field, whose value directly determines which variant is "active" in the current record. This tag field can immediately precede the variant part of the record or can be in any field whose representation is prior to that of the variant part.

Occasionally variants are discriminated based on an expression involving the values of multiple prior fields. This happens often when a record format is extended with more and more fields and subrecords over time in new versions of the software as features are added.

10.3 Nullable Types

There are many different representations of nullable data. These fall into two categories, in-band or reserved-value representations, and out-of-band or flag representations.

An in-band nullable can be implemented as a type which has one field. As an example let's consider a ZLNullString type:

```
<deftype name='ZLNullString'>
   <record>
      <field name='content' type='string'/>
      <property nullable="true"
                nullMethod="inBand"
                inBandReservedNullValue=""/>
   </record>
</deftype>
```

Now when we ask, at run time, for the value of the isNull attribute of this field, a runtime calculation occurs to test whether the reserved in-band null indication is present or not. In this case we say that strings of zero length (empty strings) are null.

One of the difficulties of in-band null techniques is the loss of some of the representation power. With a definition of ZLNullString like the above, application logic can’t distinguish the empty string from a null value. Consider a survey filled out by a WWW surfer:

How did you find our web site?

(a) Search Engine

(b) Magazine Advertisement

(c) Other ___________________________

There’s a difference between choosing the third choice, but leaving the text blank (empty string) and not answering the question at all. (null value). Because of this many kinds of data use flag represenations for nulls. Let's look at an example of null values using flag representation:

```
<deftype name='FlagNullString'>
   <record nullable='true'
           nullMethod='flag'
           nullFlagField='flagbit'
           isNull='{ ((flagbit AND 0x80 ) == 0x80 )}'>
      <field name='flagbit' type='byte'/>
      <field name='content' type='string'/>
   </record>
</deftype>
```

Here we see a very precise definition of the isNull attribute, which is computed dynamically by looking at the field named 'flagbit' within the record. The flagbit field is a single byte which precedes the content of the record. The null indicator is the most significant bit of this byte, and if it is a 1 it indicates a null value.

10.4 Using Expressions for the Value and Validity of a Field

Beyond using expressions to compute additional properties it is possible to use expression to compute the actual value of a field. Consider this example:

```
<deftype name='RatingWithQuality'>
    <record>
       <field name='data' type='int32'/>
       <property value='{ data AND 0xFFFFF }'
                 quality='{ (data AND 0xFFF) >> 20 }'/>
    </record>
</deftype>
```

In this example we started from a simple int32 field containing a metric called “rating”. It turns out that the values of ratings never need all 32 bits of the binary representation. They use at most 20 bits. Hence, there’s 12 bits left. This lets us pack an additional field into this word without having to enlarge the record format.

The distinguished property named ‘value’ behaves differently than other properties. When an expression contains the name of a field of this type, then the value property, if defined, is evaluated to produce the value of the field. In other words, an unqualified field name is equivalent to fieldname/@value, which returns the value of the “value” attribute/property.

Similarly, the distinguished property named isValid is used to determine if the value of the type is valid. For example consider this type definition for an octal digit. A string of 1 character containing digits 0-7 only:

```
<deftype name='OctalDigit'>
     <record isValid='{ isDigit(data.element(0))
                        and data.element(0) not = '8'
                        and data.element(0) not = '9' }'>
        <field name='data' type='char'/>
        <properties />
     </record>
</deftype>
```

In this example we see the definition of the isValid property which checks for the right digit values using expressions that call some simple tests and perform indexing into the string.

11. Attribute Inheritance

Default attribute values for the attributes can also be associated any point in the hierarchy.

Using deftype, one can create new types which inherit attributes from parent types in the usual object-oriented way; however, there are really two ways that an attribute's value can be inherited. The type hierarchy and the containment hierarchy. That is, if an attribute is defined in an enclosing data item, then unless there is a definition of that attribute defined for the item by its type definition, then the value of the attribute from the closest enclosing data item is used.

(This mechanism of dual inheritance has been used in a variety of software systems for many years, but has not been applied to data format descriptions before to my knowledge: Cite ICAD, Curl, etc.)

(TBD: example of exactly what we mean here.)

(TBD: explain the "default" construct, used previously in the examples.)

12. Defaulting and Error Handling

If the underlying data cannot be interpreted to create a datum of the required type, or if the restrictions are violated, then an error has been detected.

If desired, an on-error default can be provided so that a valid replacement value can be substituted. In this case the error is masked in the DFDL processor and the new value is silently provided. Applications cannot determine whether an error occurred or not.

If desired for a nullable field type, a null value can be created on error. (If multiple null values are available, then one of the null values can be specified.)

The data handling APIs need to provide control over this situation since sometimes it will not be desirable to substitute a single value.

(TBD: more about defaulting and error handling?)

13. Associating DFDL with Data

DFDL as a description is a piece of metadata. It can stand alone. E.g., it can be used to create a metadata description of data that doesn't yet exist, or that exists in separate files from the DFDL description.

Our hope is to take DFDL farther than this and to help solve the problem of keeping track of the association of metadata with the right set(s) of data. To achieve this DFDL should be able to be tightly tied to the data it describes in several ways.

- Embedded: Data in arbitrary binary format can be encoded, along with optionally being compressed and/or encrypted. The encoding can be a known standard XML encoding such as binHex or base64. The data can then live within the same XML document that contains the DFDL metadata for that data.

  The drawback of embedded data in XML is that for very large data the time needed to encode it can be very large and depending on the processing, not worthwhile. This is also not suitable for data streaming in many instances, makes random-access challenging, etc.

  Note: encoding algorithms should be well known, like base-64, or should be specified by URIs which specify the exact encoding technique used. E.g., gzip version 2.7 with a URL to the source code in ANSI C for a decompressor. This is required for the lifetime of encoded data to span multiple years.

- Associated: An XML file can contain a DFDL descriptor and a list of one or more URLs identifying the data.

  These URLs can be absolute or relative.

  The drawback of associated data is that while the XML DFDL descriptor can be used to keep everything together, the pieces can be moved around without updating the descriptor, leaving the descriptor broken, so maintaining these descriptors requires tools and discipline.

- Attached: Untouched binary data is stored in the same file with a XML DFDL prepended descriptor.

  Possibly this would use a means like the MIME “multipart” standard to encapsulate the XML DFDL and the binary data together.

  - (TBD: MIME appears to rely on delimiter strings that don’t appear within binary data. This is only reasonable if data has been encoded (e.g., base64) such that we can easily invent a delimiter string that uses characters that aren’t allowed in the encoding. Since our data is too big and too costly to encode (sometimes it is!), we need an explicit length field where the length is of unbounded precision. I.e., we need at least a 64-bit integer worth of precision. MIME allows explicit lengths to be provided only for external data. That is, data that is not stored directly into a message. So, MIME is probably not by itself usable. Alternatively, we don’t store the length at all. We just have the DFDL part, which is XML, followed by binary data until EOF. This is consistent with streamability, i.e., producing a dataset in a stream with the DFDL header first, then a variable amount of data the amount of which isn’t known in advance.)

- Referenced: A DFDL description can be referenced from a descriptor file. This is directly analogous to how an XML file refers to its XML Schema via a URI. The file can reference the DFDL descriptor and the data file(s), or one could share a DFDL data descriptor among files that reference the DFDL descriptor but contain the data directly.

Use of DFDL with these mechanisms gives much of the attractive properties of XML-format data back to legacy format data, that is, the strong association of metadata to the data, and the self-contained file formats.

14. Implied XML Schema

A DFDL data description can be automatically turned into an XML Schema that can represent the same data, albeit transformed into XML, but in such a way that the information content is not changed.

We envision the creation of tools to:

- Process binary data as if it were XML data. That is, the XML Infoset can be understood from the DFDL description of the data without having to literally translate DFDL into an XML Schema entirely. This has been called “Tagless XML”. (TBD: citation needed)
- Convert data to XML automatically and vice versa.
- Debug DFDL descriptions by use of XML tools.

The following are differences between DFDL and the model one gets back if you tranlate it into XML:

- delimiters are gone (XML tags take their function)
- all attributes about binary reps are gone.
- charset is gone, replaced by the XML charset

Some translation of character code zero within string data must be provided. E.g., use binHex.

15. Semantic Extension

DFDL should be extensible in ways that give it semantic headroom. It should be clear how to attach any of the following

- Doc (e.g., Cobol FD, system, compiler, version)
- Exact identifier of the job that created the data
- GUID (Global unique identifier) for tracking the data. I.e., identifying this data uniquely within a database.

  date, time of creation

  system data was created on (i.e., what machine and its configuration)

  userid the job ran as (or was submitted as)

  URL to the job definition, version number of that job
- Owner of the data

  might be someone to contact if you need the decryption key for the data
- Certificates indicating that the data is fit for certain purposes. E.g., that the name and address data is US Postal CASS compliant and therfore qualifies for discounted postage if used for a mailing list.
- checksums and digital signatures of the data
- field-level detail, e.g., this name address and zipcode field are the output of version 4.7 of the Integrity US Postal data cleansing module.
- sample data (in XML encoded form as part of the descriptor. Probably base64 or binhex). Drawn by some formula which is also described.
- Comments by humans. Can include arbitrary links.
- Lineage information about the origin of any field. E.g., nzipcode is derived from ozipcode and standardized with integrity version 4.3.

- statistical properties of the data. E.g., mean and std-deviation for all numbers in the data. Other statistics too. Histogram of values for all fields with <N distinct values.

Note that any of these things should be able to be embedded either all in one spot, or parcelled out through the fields. Or they should be able to be referenced via URLs instead of being embedded in the document.

16. DFDL compared to XSDL

In the same way that XML Schema Definition Language (XSDL) provides the metadata for XML data, the DFDL should provide metadata for non-XML data. XML data is very hierarchical in nature and XSDL provides mechanisms for precisely specifying hierarchical data. Non-XML data is also richly hierarchical, so DFDL must similarly provide mechanisms for its description. The relationship between the requirements on XSDL and the requirements on DFDL is so close that it makes sense for DFDL to be based very closely on the concepts of XSDL. Some concepts can be copied wholesale. The primary point of departure is that DFDL must describe non-XML data which is far less restrictive in its layout and form than XML data which, after all, obeys a strong set of rules in order to even be called well-formed XML.

16.1 Similaritites to XSDL

We expect many of the issues that come up in the creation of large complex XML schemas to have analogous issues in the creation of large complex DFDL descriptions; hence, we suggest that DFDL adopt the XML Schema Language solutions to many of these problems. Specifically, it seems DFDL should use the same solution as XSDL for these problems:

- distribution of a definition over multiple source files
- include
- import
- redefine
- namespaces and qualified names
- derived types
- extension
- restriction
- definitions and uses of reusable groups of elements
- named and anonymous types
- annotations: documentation and appnotes
- rules for equality of type and value

This document does not go into each of these topics since they are about making a useful and extensible description language for any purpose. That is they are not about the problems of describing binary and other non-XML data, which is the focus of this document.

One example is worthwhile and illustrative. Suppose some data is the output of a Cobol program and there is a Cobol file descriptor that describes the layout of this data. Suppose the program is run on an IBM mainframe machine. Now given the text of the Cobol file descriptor, the version of the Cobol compiler, and the machine type, we can construct a DFDL description of the data in the file. Hopefully tool programs will be written that perform this conversion for us; either way we may want to put the text of the file descriptor and the other details like machine type and compiler version into the DFDL as documentation.

We might similarly want to convert the output data from this same Cobol program into XML. In that case we'll need an XML schema for what the data looks like once it arrives in the XML representation. Again a program might convert this Cobol file descriptor and other information into an XML schema for us, and just as above, we may want to put the text of the file descriptor and the other details into the XML Schema as documentation.

Now, XML Schema provides a syntax for putting this documentation into the schema in a standard way. So, DFDL can simply copy this mechanism exactly. There's no reason for DFDL to not use the exact same conventions for putting this documentation into the DFDL description.

In addition, wherever DFDL uses identifiers that are the same as identifiers in XML Schema, the meaning should be the same or strongly analogous so that familiarity with XML Schema provides some degree of comfort with the style of DFDL definitions. At minimum identifiers should not have conflicting definitions or subtle differences of meaning.

For example, XSDL provides a means to express the precision and scale of a decimal number. In general properties like these are called 'attributes' in XSDL and they are used for expressing restictions on the simple types that XSDL provides. The precision and scale attributes for decimal numbers are 'totalDigits' and 'fractionDigits'. We recommend that these names be used for consistency with XML Schema rather than using the more typical legacy names 'precision' and 'scale' which have always been problematic to learn and remember. (E.g., which one is the number of fractional digits? It's quite ambiguous to a first-time user.)

16.2 Differences from XSDL

XSDL is about XML documents. The basic rules of what can be in an XML document are assumed and the strong tagged structure of XML eliminates the need for certain kinds of expressiveness. DFDL is about any data format at all. It can be binary or text or mixes of the two. In XML the lengths of data values are always delimited by an end tag. In non-XML data, text fields may be delimited by recognizable ending characters or strings, or lengths may be stored in prior fields of the record. Hence, the expressive power of DFDL must allow us to describe important relationships between different parts of the data, and must allow us to discuss characters in character sets other than the one the DFDL descriptor itself is written in.

(Note that XSDL doesn't preclude a different character set for the schema and the document. The schema could be in ebcdic-cp-us, and the document in UTF-8. The XML Information Set concept assures us that the information content is essentially character-set independent. However, in DFDL the character set can be controlled at finer granularity. It could be different on a field by field basis. This would be uncommon, but it is certainly possible.)

For example, suppose some data contains a variable length character string represented by a stored length integer prefix taking up either 1 byte or 4 bytes. The length is 1 byte in size if the string's length in bytes is between 0 and 128 bytes. If the string is longer than that the length field is 4 bytes long and is a bigendian 32 bit integer with the most-significant bit set to 1. XML can't have this kind of ambiguity in representation to resolve. There's always an ending tag which terminates things. DFDL must be able to express the relationship where one field, even one with some variant encoding rules, determines the length of another field. In order to walk the data one must extract the values of fields. This never happens in XML where it is always possible to find an element looking only at the tags of other elements, but never the contents of them. .

16.3 Record Concepts Mapped to XSDL

XSDL Concept	DFDL Concept
Element	Item
Root Element	TopLevel
Type	Type
Attribute	Attribute (also called Property herein)

Group	Group
Sequence	Struct/Record
Choice	Choice
No equivalent	Representation (text or binary)

Things in XSDL but not in DFDL include:

- 'all' groups
- final, block, abstract

  (TBD: Why? Because it seems like a bad idea in XSDL too!)
- identity constraints: both uniqueness and key constraints.
- simple content and mixed content: There's no notion of content in DFDL outside of fields. The DFDL equivalent of empty content is just a record (or subrecord) with no fields.

17. Conclusion

While the handling of commercial data descriptions is complex, it is possible to create a uniform data format description language which can, once and for all, solve the problem of describing data formats. In doing this it is necessary to acknowledge that data formats can be arbitrarily complex so that the DFDL needs to include full programming capabilities. At the same time these capabilities are not inconsistent with convenient capture and expression of most common data formats.

The ubiquity of grid computing in the future makes the GGF the ideal forum in which to advance an industry standard associated with data format descriptions.

It is hoped that the ideas in this proposal will converge with the work of others on DFDL toward a true industry standard.

18. Security Considerations

As noted herein, there are requirements when writing data for fields to be properly initialized before writing so as to prevent accidental (or purposeful) transmission of data in the unused parts of data formats.

19. Appendix: Field Length for Character Fields

Character sets are divided into two types: fixed-width and variable-width. iso-8859-1 is a fixed-width, single-byte character set. That is, each character code is represented by a single byte. UTF-16 is a fixed-width, double-byte character set. That is, each character codepoint is represented by 2 bytes. UTF-8 is a variable-width character set where a single character code can be encoded by from 1 to 4 bytes. (Note that all character sets except UTF-16 store character codes. UTF-16 stores a 2-byte character codepoint. This is an exception due to rare characters with Unicode character codes above 0xFFFF since these require more than 2 bytes to represent. See the appendix: "About UTF-16 and Unicode Character Codes above 0xFFFF".)

19.1 Field Lengths: Fixed-Width Charsets

When a field is specified to have text representation, then these attributes are used to specify and bound the length:

- length
- minLength
- maxLength

The units are character codes (or 2-byte codepoints for UTF-16). From these the values of the attributes lengthInBytes, maxLengthInBytes, and minLengthInBytes can be computed without having to see the actual data, that is they are statically known just from the DFDL description. Again, this is for a fixed-width charset.

> (Note: We could actually strengthen this. For text fields where all allowed values are enumerated we could compute the max and min byte length even for a variable-width character set like UTF-8. This is probably not worth it, and may get in the way of creating diagnostic validating DFDL data parsers. Note that this problem generalizes to computing the length or maxLength in bytes for a string where a regular expression pattern constrains the specific characters and where they can appear. Hard to tell if this problem is actually computable theoretically!)

It is not generally the case that one can determine length from other attributes. That is, one might think that for numeric types an attribute like totalDigits can be used to determine the length; unfortunately this is generally not true for textual representations without introducing many other attributes which control in detail every aspect of how numbers can be formatted (whitespace allowed, comma separators required, sign required or optional, etc.) This complexity isn't worth it.

19.2 Field Lengths: Variable-Width Charsets

For variable-width character sets, a text representation can have a fixed number of characters, but the representation of that string can take a variable number of bytes; hence, fields using variable-width character sets, but which specify the length attribute are not fixed length fields in bytes, they are variable length in bytes.

> (Note: We could strengthen this. For text reps for number types we know the width of each digit and any other valid character that can appear in the rep of a number; hence, we can compute the exact width even if the charset is variable-width because there are no such character sets where the various digits are differing widths. The most interesting example here is UTF-8, where all the valid characters for numeric types are 1 byte wide. In general this capability is probably not worth it, and may get in the way of creating diagnostic validating DFDL data parsers.)

When a field has text representation and a variable-width character set, then its lengthInBytes is not knowable from the metadata, although its minLengthInBytes and maxLengthInBytes may be bounded by making worst-case assumptions about the characters that could appear. For example, if the length is 10 then the worst case lengthInBytes for charset UTF-8 is 40 bytes. The minLengthInBytes is 10.

When a field has a text representation and a variable-width character set, and the length attribute is specified with an integer value, then the length in bytes can be computed at run time by parsing the text from start towards the end. (This parse can be nearly as efficient as searching for a fixed delimiter. Today's processors don't have hardware instruction sets for it, but the loop is very very tight for charsets like UTF-8.) When the length number of character codes have been found, that identifies the end of the variable-length representation.

19.3 Field Lengths: Summary

Since we can determine the length in bytes by scanning the string in the variable-width case, this assures us that for all primitive data types, we can parse the data and identify the position of every field so long as we have:

- lengthInBytes for binary fields, or totalDigits for decimal binary fields .
- Charset and length for text fields.

If all charsets are fixed-width then a record containing all fields as above is fixed length, and all fields begin at fixed positions within it that can be statically computed.

If any charset for a field is variable-width, then we must determine the length of that field at run-time by examining the data.

Example: Consider the string '1.23E45.678' as two doubles with text representation. How do we determine the boundary between them. It's ambiguous in any charset. We either need a delimiter explicitly, a whitespace character (which is a kind of error-prone delimiter) or the length.

In summary: you can still think about characters as if they were fixed-width and think about fields with specific numbers of characters in their representations as if they were fixed-width. Functionally, it will work. There is a performance penalty for variable-width charsets since field positions are no longer statically computed.

It is desirable to be able to ignore/suppress the variable-width aspect of some charsets. For example, changing charsets for a text-representation of a field from ebcdic-cp-us to UTF-8 does not require you to add in new aspects to your data such as delimiters between fields.

20. Appendix: About UTF-16 and Unicode Character Codes above 0xFFFF

When we define UTF-16 to be a fixed-width double-byte wide character set we say that each UTF-16 codepoint is represented by 2 bytes. Notice the careful use of the term 'codepoint' here. Unicode characters can have character codes as large as 0x10FFFF which requires 3 bytes to store (21 bits actually); however in UTF-16 characters with more than 2 bytes of code are encoded as two codepoints, called a surrogate pair; hence, UTF-16 is fixed-width, 2 bytes per codepoint. It is not really 2 bytes per Unicode character code. UTF-16 is really a variable-width encoding, but the characters that require the surrogate-pair treatment are so infrequently used that UTF-16 is most often treated like a 16-bit fixed-width character set. It is the acknowledgement of the existance of surrogate pairs that leads to the "codepoint" vs. "character code" distinction.

UTF-32 is a fixed width encoding with a full 4-bytes per character code. It represents all of Unicode with the same width per character.

Hence, when we refer to positions or lengths in character strings we will often refer to length in chars, but always qualify that it means 2-byte codepoints in UTF-16.

21. Appendix: About Literal String Values in DFDL

In the XML Schema of the DFDL language, wherever a string may be specified which is to be compared to the external data being described by the DFDL description, we don't specify that the XML data is an ordinary string, but rather we specify a charset, and then express the data in that charset. However, as DFDL is an XML-based language, we're expressing the characters of this charset in the character set specified in the XML declaration of the DFDL file. Certain character codes are disallowed in XML, yet we need to specify those character codes in delimiters, patterns, enumerations of allowed values for restricted types, and so forth.

The character with codepoint 0 is not valid in any XML 1.1 document (XML 1.0 disallows several character codes). So, it is not possible in the XML syntax of a DFDL descriptor to write '&#x00;' or '&#0;' or any other way to actually embed the character code 0 into the DFDL descriptor document.

Nevertheless, there are lots of examples where we need these character codes. We need a way to specify that the character with character code zero is to be used as a delimiter string. This is a common data convention, C programs frequently use ASCII NUL terminated strings, and the character code zero is certainly a valid character in much data.

Hence, a meta-escape is required. This meta-escape is needed for XML 1.0 for several characters, and for XML 1.1 only for the character code 0.

For example, suppose our DFDL descriptor is itself a UTF-8 document, but we're specifying a delimiter string in the ebcdic-cp-us charset because that's the character set that our data uses. The characters that appear for the delimiter in our DFDL document are UTF-8 encoded. These characters must be translated into the corresponding characters in the ebcdic-cp-us character set by a translation from utf-8 to ebcdic-cp-us, presumably going to full unicode from utf-8 along the way. This works fine for everything except a few character codes in XML 1.0, and the single character code 0 in XML 1.1.

To encode the missing values we allow all literal strings in DFDL to be specified optionally using a hexBinary encoding. This is specified by a new attribute:

- stringLiteralEncoding: values are hexBinary or none. None is the default and indicates that string literals are represented in the character set of the DFDL XML document. The hexBinary representation is of the characters as encoded in the target charset. I.e., With a stringLiteralEncoding of hexBinary and a charset of ebcdic-cp-us, the literal '4E' represents the character '+' in the ebcdic-cp-us encoding, not the character 'N' in UTF-8 encoding of the DFDL descriptor itself.

(TBD: Perhaps we should support other encodings more convenient to humans? E.g., the RFC 2396 encoding where a-ZA-Z0-9/:. and hyphen are represented by themselves but other characters must be escaped by a %hh convention where the %hh turns into one byte of a UTF-8 encoding of the character code. That's one solution, but a more Unicode friendly convention might allow escape sequences like \Uhhhhhh or \U{h..h} with 1 to 6 characters to specify a codepoint.)

(TBD: wouldn't it be nice if based on our needs for DFDL we could get the XML 1.X people to lift the darn restriction on character codes and just allow '&#0;' to work?)

22. Appendix: Decimal Number Attributes Detail

There are two approaches to dealing with the complexity of decimal representations. One is to enumerate and separately name every combination that is actually in use. This results in a list of hundreds of different formats, the differences between them are subtle. The other is to tease-apart the different varying aspects that make up the different formats. This has the drawback that many formats can be described that have never existed in history. Nevertheless, we've chosen the latter approach. Using the extensible types in DFDL new types can be defined which correspond to certain frequently occuring standard combinations of these options. This gives us the best of both approaches.

The attributes are:

- packed: This is a binary format which indicates that the data is stored as two decimal digits per byte. If the number is signed, one nibble is used to store a sign nibble. If the number is unsigned no nibble is used for sign and it is instead used for a digit. For input, hex nibbles A, C, E, and F indicate positive, while B and D indicate negative. For output C should be generated for positive and D for negative.
- Cobol COMP-3 data is packed decimal.
- packedPlusSign: Specifies a hex nibble to use for the plus sign when writing out signed packed decimal numbers. Defaults to 0xC. Could also be used to validate input data having a specific sign convention.
- packedMinusSign: Specifies a hex nibble to use for the minus sign when writing out signed packed decimal numbers. Defaults to 0xD. Could also be used to validate input data having a specific sign convention.

It is possible for an extra nibble to be unused in the representation of a packed decimal number. For example, if there are 4 digits plus a sign, then 5 nibbles are needed. When 3 bytes hold these nibbles, one nibble is unused. When writing data out, this nibble must always be zeroed for security reasons.

(Note: About unsignedDecimal RM/COBOL COMP-3 UNSIGNED data has a 0xF sign byte anyway. So it would be described in DFDL as signed, i.e., conversion of a Cobol FD for a RM/COBOL system would translate COMP-3 UNSIGNED into a signed packed decimal description in DFDL. Source: MFCobol Conversion document cited elsewhere herein.)

- unpacked: each digit is stored as the least significant nibble of a byte. The most significant nibble is always zero. The sign nibble is as for packed decimal but is also stored in the least significant nibble of an additional byte (for signed numbers). Note that unpacked is not a text format. In most character sets the character codes like 0x05 are non-printing characters. In XML the Unicode character with code 0x00 isn't even allowed (as of XML 1.1). This makes the unpacked representation really an infrequently used binary format.

  Note: unpacked is used by RM/Cobol (see MicroFocus Cobol Compatibility Guide, April 1993, Chapter 8: "Converting RM/COBOL Data Files", page 8-1.) (TBD: need better citation)

(TBD: packedPlusSign and packedMinusSign apply even to unpacked representation. RM/Cobol uses different defaults than IBM mainframe Cobol. 0xB is the plus sign. So, for unpacked data should these conventions just be different?)

- tolerateZeroSign: A common data convention is that if all bytes are zero then the number should be assumed to be zero even though technically it's not valid packed (or unpacked) decimal unless it has a positive sign (A,C,E, or F) or negative sign (B or D).

  (TBD: citation - this comes from Ascential Orchestrate manuals where it is the 'fixzero' import/export property)

- signOverpunched: true, means sign is overpunched. False, or separate mean the sign is separate: For text or unpacked representations this determines if the sign is encoded on top of an end digit. This format saves a byte for a separate sign, but lets things like '12{' be valid decimal numbers. The default is separate for all text representations and the default is overpunched for zoned binary representation. The valid characters used for overpunched signs vary depending on the character set.

- zoned: each digit is stored as a byte. The high nibble contains 0xF for EBCDIC charsets and 0x3 for ASCII-based charsets. The sign is leading or trailing, and is usually overpunched. If overpunched, the same nibble values are used as in packed decimal representation. If separate, then bytes 0xC0 and 0xD0 can be used to represent the plus and minus signs respectively, or the characters for '+' and '-' in the character set. Note that 0xC0 is the '{' character in EBCDIC and 0xD0 is the '}' character in EBCDIC. Because of the likelyhood of flawed translation of zoned decimal data from EBCDIC to ASCII, the ASCII byte 0x7B and 0x7D should also be accepted as specifying the plus and minus signs respectively.
- The only difference between zoned decimal and a text string representing the number is that zoned decimal has an implied decimal point location specified by fractionDigits, so periods, commas and other textual characters are not allowed in the representation. More specifically, a zoned decimal number can contain only these characters '0123456789+-{}', in EBCDIC or ASCII characters with 1 byte per character where the braces correspond to plus and minus signs.
- Zoned decimal fields are a binary format that is sensitive to character set. On output a zoned decimal type field contains EBCDIC characters if the charset is any EBCDIC variety and ASCII single-byte characters otherwise. More specifically, if the charset is UTF-16, a zoned decimal field still contains one byte per digit, with zero represented by 0x30. We do not represent each digit of a zoned decimal number by a two-byte character code just because the charset is UTF-16.

22.1 Example: Decimal big-endian, zoned, trailing overpunched sign

Let's consider this example using the EBCDIC character set (example from http://www.discinterchange.com/TechTalk_signed_fields_.html) :

This is how the value +123 would be stored with the sign in the LSD (least significant digit):

- The 1 is stored as the EBCDIC character "1", which is 0xF1.
- The 2 is stored as the EBCDIC character "2", which is 0xF2.
- The 3, however, is converted to a binary value of 3, which is 03 hex, then OR'd with the positive sign code of 0xC0, resulting in the byte 0xC3.

The resulting field in hex is: F1, F2, C3. The proper way to interpret the C3 is as two entities: the sign of C0 and the value of 03. However, it also happens to be the code for the letter "C", so if you view the field in EBCDIC character mode, you will see "12C".

The value of +120 would likewise become F1, F2, C0 (00 OR'd with C0). Since the character assigned to the value C0 is a left brace, "{", when viewed in EBCDIC character mode, you will see "12{".

22.2 Overpunched Signs in ASCII Data

The exact behavior, i.e., what characters are used to represent overpunched positive and negative digits, varies by character set and Cobol compiler and system. By far the most common convention is the EBCDIC one described here used on IBM mainframe systems.

In EBCDIC character sets, overpunched characters '}JKLMNOPQR' represent a negative sign and digits 0 to 9. The characters '{ABCDEFGHI' represent a positive sign and digits 0 to 9.

In ASCII character sets, there are several conventions.

- EBCDIC converted style: The overpunched character is the ASCII equivalent of the EBCDIC above. In other words, characters '}JKLMNOPQR' still represent negative sign and digits 0 to

9 even though when looking at the bytes in hex this makes no sense whatsoever because the "}" character has ASCII code 0x7B.

- MicroFocus and Microsoft Cobol Style: ASCII characters 'pqrstuvwxy' represent negative sign and digits 0 to 9. ASCII characters '0123456789' represent a positive sign and the corresponding digit. (Sign nibble for '+' is 0x3, which is the high nibble of these character codes unmodified.)
- CA Realia Compiler Style: ASCII characters 'pqrstuvwxy' represent negative sign and digits 0 to 9. ASCII characters from code 0x20 to 0x29 are used for positive sign and the corresponding decimal digit. These characters include the space (' ') for zero, characters '!"#$%&' for 1 through 6, the single quote character "'" for 7, and the parenthesis '()' for 8 and 9.

So, in summary...

If the character set is EBCDIC (or any variant thereof) then bytes with high nibble of 0xF are just digits. Bytes with high nibble of 0xC indicate a plus sign and a digit, and 0xD for minus sign and a digit.

If the character set is ASCII (or any variant thereof) then bytes with a high nibble of 0x3 are just digits. For a signed number a byte with high nibble of 0x3, or 0x2 is a plus sign and a digit, a byte with a high nibble of 0x7 is a minus sign and a digit. The byte 0x7B (character '{') is a plus sign and the digit 0 (zero). The bytes 0x41 to 0x49 (characters 'ABCDEFGHI') are a plus sign and the digits 1 to 9. The byte 0x7D (character '}') is a minus sign and the digit 0. The bytes 0x4A to 0x52 (characters 'JKLMNOPQR') are a minus sign and the digits 1 to 9 respectively.

This means that for input of overpunched sign decimal numbers, a single algorithm suffices to interpret all known (?) encodings unambiguously without any need for users to specify or know more than the character set being used.

Unfortunately, when writing out this overpunched sign representation, if the character set is EBCDIC or a variant, then the convention is unambiguous. If the character set is ASCII, then one must specify a value for a attribute to control what convention is used for the sign encoding.

- overpunchedASCIISignStyle: one of ebcdicConvertedStyle, MFMSStyle, CARealiaStyle must be specified. On inputs this could also be used for input validation if you know the style that the numbers should be in.

(TBD: any default? The default should be selected based on what style is found when data is read; however, what if a program isn't doing any reading? Then the default really is ambiguous. For a flag this obscure it is unpleasant to make it required. My guess is that ebcdicConvertedStyle is the most common, but that's a guess. -mikeb 2003-08-20)

(TBD: other conventions may be out there. So, this set may get extended over time. This set is all I could find in my own documentation and by searching the Internet with Google on 2003-08-20 - Mike Beckerle, Ascential Software)

The SAS Language Reference describes the 'Informats' accepted by SAS. These are a quite comprehensive list of data formats for decimal and date types. This document is not provided online by the SAS Institute, but many organizations have put it online for their users. One such source about the 'Informats' topic is http://www.ncsu.edu/it/sas/help/lrcon/z0920449.htm .

23. Appendix: Ambiguity of Date/Time Information

Consider this example field: “06/07/08 24:19:11.876”. The “06/07/08” is presumably a date, but what parts are what? Let’s assume we determine that the format is “DD/MM/YY”. It’s now ambiguous what year was intended by the “08” is that 2008, or 1908, so we need to know what year basis is used to disambiguate 2-digit years. Let’s assume it was 1908. A couple of other things are also ambiguous, though defaults are probably correct for most commercial data. These are the calendar being used, and the era. The calendar is probably Gregorian if the application is commercial; however, the last country to convert to the Gregorian calendar was Turkey (1926-01-01) prior to that Greece (1924-03-24) (source www.ssa.gov). Clearly there are many people still alive today whose birth dates are ambiguous because of this. The era is certainly CE (a.k.a., AD) and this should be the default setting; however, it should be possible to be specific, and there should at least be future extensibility to other calendars and their eras.

What about the time part “24:19:11.876”. Even if we assume the notation is hours, minutes and seconds, is this value 24 mean this time is invalid or is a 1-based hour-numbering being used here. (Some places/people count hours starting from 01:00 to 24:59.)

The description of date and time features involves many string literals in the DFDL description. See the appendix "About String Literal Values".

Date and time formats vary widely from one locale to another. Months and days of the week have different names of course, with a variety of accepted abbreviations, but the difficulties go further: For example, in Italian, the era which in English is notated “AD” or “CE” is notated “DC” (D for “dopo”), and the era notated in English as “BC” or “BCE” is notated “AC” (A for “avanti”). Of course either the English or the native Italian language form could be used.

Note also that 60 is valid for seconds, not just 0 to 59. This is not of commercial importance, but it is needed to accommodate leap-seconds which are added every few years. For example, the following is a valid ISO datetime: “1998-12-31T23:59:60+00:00”.

23.1 Output Formats for Dates and Times

Certainly for all data being created today, and for representing all dates and times since 1926-01-01, ISO 8601 textual formats are the recommended representation. For example these are two ISO 8601 date and time strings:

1926-01-01T00:00:00.000+00:00

1926-W01-1T00:00:00.000+00:00

This is the first truly unambiguous millisecond of time! All times prior must be qualified as to what calendar applies.

24. Appendix: Date, Time, and Duration Patterns

This stuff is straight from ICU. With the exception of the Kanji examples using XML entities.

The date/time and duration textual format is specified by means of a string pattern. The count of pattern letters determines the format. In this pattern, letters are reserved as pattern letters:

Symbol	Meaning	Presentation	Example
G	era designator	(Text)	AD
y	year	(Number)	1996
M	month in year	(Text and Number)	July and 07
d	day in month	(Number)	10
h	hour in am/pm (1~12)	(Number)	12
H	hour in day (0~23)	(Number)	0
m	minute in hour	(Number)	30
s	second in minute	(Number)	55
S	millisecond	(Number)	978
E	day in week	(Text)	Tuesday
D	day in year	(Number)	189
F	day of week in month	(Number)	2 (2nd Wed in July)
w	week in year	(Number)	27
W	week in month	(Number)	2
a	am/pm marker	(Text)	pm
k	hour in day (1~24)	(Number)	24
K	hour in am/pm (0~11)	(Number)	0
Z	time zone	(Text)	Pacific Standard Time
'	escape for text		
''	single quote		'

The ‘Presentation’ is interpreted as follows:

- Text: Four or more, use full form, <4, use short or abbreviated form if it exists. (for example, "EEEE"" produces " Monday", "EEE" produces "Mon" in the US locale. In the FR locale, for the same value, “EEEE” produces “Lundi”)
- Number: The minimum number of digits. Shorter numbers are zero-padded to this amount (for example, if "m" produces "6", "mm" produces "06"). Year is handled specially; that is, if

the count of 'y' is 2, the Year will be truncated to 2 digits. (for example, if "yyyy" produces "1997", "yy" produces "97".)

- Both Text and Number: Three or over, use text, otherwise use number. (for example, "M" produces "1", "MM" produces "01", "MMM" produces "Jan", and "MMMM" produces "January". The text chosen is sensitive to the locale selected.)

Note: (From the ICU web site) Any characters in the pattern that are not in the ranges of ['a'..'z'] and ['A'..'Z'] will be treated as quoted text. For instance, characters like ':', '.', ' ', '#' and '@' will appear in the resulting time text even if they are not enclosed within single quotes.

Format Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss Z"	1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, ''yy"	Wed, July 10, '96
"h:mm a"	8:08 PM
"hh 'o''clock' a, ZZZZ"	09 o'clock AM. Eastern Standard Time
"K:mm a, Z"	9:34 AM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"	1996.July.10 AD 12:08 PM
"yyyy年mm月dd日"	2003年08月27日
"yyyy&#x5e74;mm&#6708;dd&#65e5;"	2003年08月27日

Note in the above the expansion of Unicode character entities is specifically allowed.

(TBD: add pattern support for quarters? As in syntax like “2Q2004”)

(TBD: add pattern support for timezone numerically as in “00:00:00+06:00”, need a way to specify that we want the “+06:00” not “CST” or “MDT” or “Central Standard Time”. The pattern letter ‘z’ might be used for this.)

25. Appendix: Record Blocking Schemes Detail

Record blocking schemes largely evolved on IBM-style mainframes. The JCL (Job Control Language) used there has a DCB (Data Control Block) which has 3 important parameters. RECFM, LRECL, and BLKSIZE. BLKSIZE normally ranges from 1 to 32760, LRECL similarly, but for some record formats it must be smaller than the BLKSIZE by some number of bytes. Finally RECFM can take on the following values: U, F, FB, V, VB, VBS. Generally, F stands for fixed-length, V for variable-length, B for blocked, and S for "spanning", so VBS means variable length records that can span blocks.

(TBD: any other schemes that are important?)

The authoritative source for the definition of the RECFM information is chapter 21 of IBM OS/390 DFSMS: Using Data Sets (IBM publication SC26-7339-01, Second Edition, December 2000).

This is available online at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DGT1D411/CCONTENTS?SHELF=EZ239126&DN=SC26-7339-01&DT=20001014144419

Other useful links to abbreviated discussions of blocking can be found at:

- http://www.system.missouri.edu/itss/ss/guides/jcl/dddcb.html
- http://www.eits.uga.edu/hostsys/mvs/ibm-jcl-intro.html
- http://www.cray.com/craydoc/20/manuals/S-2351-50/html-S-2351-50/fixedg6tdnebl25.html

26. Appendix: About Text Floating Point Numbers and Rounding

When a floating point number is represented as a text in base 10, it is possible for it to be ambiguous what binary floating point number is to be chosen as the representation. That is, it is possible for two different binary floating point numbers to be equidistant from the ideal base 10 number. Both binary numbers if converted back to base 10 would have the exact same base 10 digits; hence, it is necessary to have a attributes to control which floating point binary number is to be used. The rounding modes are either ceiling, floor, round_inf, trunc_zero which mean to round up, round down, round toward closest infinity, or round toward zero.

(TBD: specific attribute names for this control)

In the opposite conversion direction (binary to base 10) there's an additional rounding mode: round_even which rounds to the base 10 digit that is even. This equally distributes the rounds with some going up to an even number and some going down to an even number.

27. Appendix: About DIME Format: Why not DIME?

DIME has been suggested with respect to understanding its relevance for combining binary data with formatting information in the same file/stream.

For those not familiar, DIME is defined in the draft IETF submission as:

> Direct Internet Message Encapsulation (DIME) is a lightweight,
> binary message format that can be used to encapsulate one or more
> application-defined payloads of arbitrary type and size into a
> single message construct. ...

Here's the URL http://www.ibm.com/developerworks/webservices/library/ws-dime/ perhaps there is a more official one, but this one is the best I found via google.

Unfortunately, DIME is flawed for our purposes:

Problem: The data length field in DIME is a 32-bit unsigned integer. Not big enough for large data. You can carry data larger than that in Dime, but only by breaking up the binary data into multiple Dime "records", each having a maximum 32-bit length, and with each record payload padded to a multiple of 4 bytes.

People with flat file data don't want to break up large data into multiple chunks this size. That often requires an extra pass or modifications to software that is producing the data, makes it impossible to directly memory map files bigger than what this 32-bit length allows, etc. So DIME isn't something one can "wrap around" big data without having to potentially modify that data. To me this is an important requirement.

The fix to this would just be to make the data length be a 64-bit integer, or preferably a textual integer (string of UTF-8 digits 0-9) which is allowed to be of arbitrary length.

Problem: DIME is a binary format header. It is completely and precisely specified as to byte ordering issues, but it's not human intelligible if you just open a file containing DIME data. This is not very modern. I would prefer something where even if it's a binary header the first 6 bytes are guaranteed to be the Unicode UTF-8 encoded characters for "<DIME>" as a human readible textual sentinel. Ultimately, I'd have prefered the entire header be text characters even if very rigidly specified as to character set, position, and so forth. There's also so much text in the DIME headers for URIs and registered type names and such that having the header be binary format seems pretty silly really.

I would not recommend that DFDL-WG adopt DIME as the means of attaching formatting information to binary data for these reasons. The first problem above is fatal. The second, we could have traded that one off, but since the first one is fatal we have to do something else so we might as well not have the 2nd problem.

Good things to copy from DIME are:

- streamability: one should also be able to output data without having to buffer it all to compute its final length.
- The DIME solution to this is ok, but we also need the ability to have the final chunk of a file/stream be binary data of unknown length so that one can write all header information then write out a massive amount of data without buffering at all, with end-of-file/stream being the implicit indicator for where the data ends. This allows the special case of a 2-part message where the first part is the data format information, and the second part is an enormous set of data, far too big to buffer, and that you want to deliver this whole thing down an operating system pipe to some process; hence, you have no choice but to stream it without knowing the length.
- no whole-file header/trailer: There is no DIME header or trailer which must preceed/follow the DIME records in between. A DIME message is just a sequence of DIME records. The record has a flag bit to indicate whether it is first or last. It can be both in the case of a message

consisting of just one DIME record. Specifically, the number of DIME records in the message isn't recorded anywhere. This also supports streamability.

Author Information

Mike Beckerle, mike.beckerle@ascential.com, Ascential Software, Inc., 50 Washington St. Westborough, MA 01581, USA

This document is provided by Ascential Software, Inc. (www.ascential.com). Ascential supplies the Enterprise computing market with integration solutions including high-volume data processing systems that span both batch and real-time computing environments. Ascential's software has run on large-scale clusters (100+ computers) since 1995 solving complex data processing problems across industries from airlines to web-target-marketing. Ascential's software emphasizes construction of flow-based software using reusable components, a set of techniques that are very effective but relatively unknown in the Grid and HPC communities today. It provides scalable parallel pathways to and from most Enterprise computing data stores including file systems and relational databases. Pluggable data interfaces are crucial to the success of this style of software application development and this drives Ascential's interest in the DFDL-WG.

References and Notes

- OMG Common Warehouse Metadata (CWM) (http://www.omg.com/cwm)

- Orchestrate Schema Format (http://www.ascential.com/) - Not publicly available.
- Mercator Type Trees (http://www.mercator.com/) - Not publicly available.
- BinX: http://www.epcc.ed.ac.uk/~gridserve/WP5/Binx/ or http://www.edikt.org/binx
- HDF: http://hdf.ncsa.uiuc.edu/
- XML 1.0 http://www.w3.org/TR/REC-xml
- XML 1.1 http://www.w3.org/TR/xml11/
- Unicode (now at version 4.0) http://www.unicode.org/
- IANA character set encoding names: (http://www.iana.org/assignments/character-sets)
- XML Schema: http://www.w3.org/XML/Schema
- MIME: http://www.faqs.org/rfcs/rfc2045.html
- ISO COBOL standard: http://www.cobolstandard.info/wg4/wg4.html
- MicroFocus Cobol 1993 Manual (http://www.microfocus.com/)
- SAS: http://www.sas.com/
- POSIX/Unix Locale http://www.opengroup.org/onlinepubs/007908799/xbd/locale.html
- ICU: International Components for Unicode http://oss.software.ibm.com/icu/userguide/index.html
- ISO 8601 standard for dates and times: http://www.iso.ch/iso/en/prods%2Dservices/popstds/datesandtime.html
- The U.S. Social Security Administration Gives date of conversion of Greece (actually the Orthodox Catholic Church in Greece) to the Gregorian calendar as 1924-03-24. (see http://policy.ssa.gov/poms.nsf/lnx/0200307180). Turkey converted in 1926, but they converted the days in 1917 and converted the year in 1926.
- Leap Seconds: see http://tycho.usno.navy.mil/leap.html or search the web!
- Java Internationalization (ISBN 0-596-00019-7)
- SAM homepage http://collaboratory.emsl.pnl.gov/docs/collab/sam/
- XSIL homepage http://www.cacr.caltech.edu/SDA/xsil/
- XDR (IETF RFC 1832) http://www.faqs.org/rfcs/rfc1832.html
- ASN.1 http://asn1.elibel.tm.fr/en/standards/index.htm
- DIME http://www.ibm.com/developerworks/webservices/library/ws-dime/
- IBM OS/390 DFSMS: Using Data Sets. IBM publication SC26-7339-01, Second Edition, December 2000. (online at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DGT1D411/CCONTENTS?SHELF=EZ239126&DN=SC26-7339-01&DT=20001014144419)

ToDo List

- Fix UML - use stereotypes for multiplicity of attributes: <<0..*>>.
- Consistency of "item" vs. "field".
- Lots of examples.
    - Lookahead – example/use case where you have header, body+, trailer where the only way to unambiguously identify the trailer is that it is the last line before EOF. Therefore processing body records requires that you look ahead to be sure that the record is not followed by an EOF, but has a line ending with more content past it.
- Defaulting and Error handling - This includes default values and when to substitute them based on error situations. Whole topic of parse failures, unexpected values, validating proessors, etc.
- polymorphism section
- Character set maps: so you can specify an exact translation
    - So we XML-encode the ICU character maps (or the Java character encoding tables from Java 1.4 NIO)
    - Probably we give a URL to a map.
- Collation sequence specifications: thse are needed as soon as you add interesting stats or metadata to the file. E.g., if you say partitioned by X, then you are specifying equivalence classes, if on string key field then collation sequence must be taken into account. Similarly, histograms of distribution for strings are based on equality comparisons of strings. If you say ordered by Y, then obviously the question arises of collation rules if Y contains any strings.
    - So we XML-encode the ICU collation sequence expressions.
    - Probably we give a URL to a collation sequence spec.
- Mulitple choices for delimiters, quoting brackets, date/time format specs.. I.e., a record where the separator is either comma or pipe. Where fields are surrounded by either [] or “”, and identical open-close quotes vs. symmetric open quote and close quote which are different characters.
- initiators - add attributes to diagrams and explanations for them. Also, should we have these for binary data or just text? Also, relationship of quote support to intiators-terminators.
- DFDL is a specification language for an attribute-grammar-based intelligent parser. Find citation to the OO attribute grammars stuff?
- Cite EDI formats: swift, rosettanet, etc. (The non-XML ones)

Minor items:

- Calendar specs: by URI hopefully. Issue of start-dates for calendars. I.e., Gregorian calendar, but starting in 1924-03-24, for Greece, not 1582-10-15, as for Rome.
- TBD: tape labels. Used to be a mainframe tape cartridge would get put into a unix tape drive and the bytes coming off would begin with the mainframe tape label. The tape image including the tape label would get put into a file, and that would be the input data. So knowing the exact formats of tape labels so you could skip past them (or even use them) was important at one time.

- standardized and end-user defined entity definitions: particularly valuable for treating non-ascii characters in ascii. E.g., Japanese Year = &#x5e74; Month=$#x6708; Day=#x65e5. Note that there are no standard Unicode names for these characters that are meaningful. They are just Unicode-CJK-Unified-Ideographs-character-5e74, or similar.
    - User defined entity definitions would let people put definitions like JYEAR=&#x5e74;
    - using DTDs with DFDL to define standard entities. (for &CRLF; for example)

This lets formats like: 2003 年08月27日 be expressed for dates easily without having to remember numeric values for Unicode characters: (This one specifically means 2003 Year 08 Month 27 Day).