DFDL Wish List

Based on experience creating DFDL schemas and running them using Daffodil, there are a number of things not currently described in the DFDL v1.0 standards documents that would be helpful in modeling.

They are described here in no particular order, along with some discussion of them.

DFDL v1.0 was designed to standardize the behaviors of many ad-hoc product-specific data format description capabilities.

One set of goals for DFDL v2.0 will be to advance the state-of-the-art, and describe data formats that none of the prior-generation ad-hoc product-specific data tools have been able to address effectively.

The DFDL Workgroup of the Open Grid Forum also has been saving some issues targeted at DFDL 2.0.

Recursion

One of the first things people want to model in DFDL always seems to be a binary legacy document formats like RTF or older MS Word documents. These have recursive structures where a section can contain text and other sections. DFDL v1.0 was not designed with document formats in mind, but rather with more traditional "data sets" or files of data in mind.

One advantage of DFDL without recursion is that it is not a 'Turing complete' language. This is helpful from a security perspective. Adding recursion may break this boundary. Of course since DFDL has a rich regular expression capability, and its own backtracking, one can still create schemas that take absurdly long to execute even without recursion, so maybe this is not an issue.

Layering - Data Source/Target Indirection

(This has an initial implementation now. The API may still evolve.)

The layering feature of Daffodil needs to be extended to enable new external layer transforms to be added via external jars.

In addition, one needs the ability to verify (parsing) or compute (unparsing) checksums, CRCs, or parity elements based on the contents of a layer.

Complex Representations for Simple Types

XML Schema's simple vs. complex type distinction is quite painful. Often you want the logical result to be some computed element (using dfdl:inputValueCalc) of simple type, but one must have a hidden sequence group of several elements that are the more complicated representation details. In DFDL v1.0, one must of necessity model such a thing as a complex type, so that you have a place where both a hidden group and the 'value' element of simple type can live side by side.

A means is needed to embed a hidden group within the definition of a simple type, so that the hidden group is implicitly laid down next to the element having that simple type.

XML Attributes

The ability to have data of simple type become XML attribute values would go a long ways to making DFDL-created-XML more human-friendly, and more efficient - attributes don't repeat the attribute name twice. E.g., <bigLongName>0</bigLongName> becomes bigLongName="0". This is very close to the same level of density as JSON representation.

More XML Schema Constructs

Several things in XML Schema seem to be missing. Unless there is a clear reason not to support them, it would be helpful. This list includes at least:

1. repeating sequence and choice groups (minOccurs and maxOccurs)
2. complex type derivations
3. attributes (already mentioned above)
4. substitution groups - to enable separate compilation of multi-part DFDL schemas that are very large. (might be overkill - unclear if this is truly needed.)

XML Schema 1.1 / Schematron

(Schematron is now implemented. Rules can be separate or embedded in the schema.)

This new standard supports richer validation rules. They are useful since XML Schema 1.0's validation capabilities are so limited.

Alternatively, embedding schematron rules directly in a DFDL schema is an option.

Delimited by Next Item

The ability to say that an element or group is delimited, but that it is delimited by the boundary of finding the initiator of the next element or group would simplify the description of many formats.

Another formulation would be to specify that an element/group is delimited, but that the terminating markup is not consumed, and hence, must be consumed by whatever comes next in the model.

Character Class Entities

(now DAFFODIL-2720 - New Char Class Entities: LSP LSP* LSP+ SP* SP+ REOPENED)

We badly need an entity that means 'any whitespace that is not a line ending'. This avoids the specification of separators like:

```
dfdl:separator="%SP; %SP;%SP; %SP;%SP;%SP; %SP;%SP;%SP;%SP; %SP;%SP;%SP;%SP;%SP;"
```

which is sometimes needed when %NL; is the terminator and you want to distinguish the separator and terminator. The %WSP+; entity encompasses all whitespace.

A possible good name is %LSP*; where the "L" is for "Linear" as in "within a line", meaning specifically matches tab or space characters. (Exactly U+0009 and U+0020, not other Unicode space-like characters)

5 new entities are actually needed: %LSP; %LSP*; %LSP+; %SP*; %SP+;

Summary Functions/Operations

For both parsing and particularly for unparsing, one often must measure something. Fixed length formats often have tabular layouts in them, and the widths of the columns need to be computed from the longest string in the data.

This is a form of multi-pass (aka Layers), but for the unparsing case, it's really just about computing the length of something from values in the infoset, a capability DFDL already has. The need is to just generalize the calculation capability with some sort of map/reduce on arrays.

Extensions with User-defined Functions

(This is implemented now.)

Security Features

No Network Mode: (This is implemented now.)

Regular Expression Enhancements

DFDL schemas involve some large and complex regular expressions. Even the most advanced regular expression languages lack convenient ability to define a given construct once and name it, and then reuse it by somehow referencing that name. This would dramatically ease construction of regular expressions, and it is simply basic software engineering that large and complex things need to be named and reused, not duplicated.

A coherent proposal here would be very useful outside of DFDL/Daffodil.

Graph of Nodes and Edges Data

Users have requested a way to describe data structures with pointers linking objects to other objects. E.g. the arbitrary link structure a typical *NIX file system image can contain, with its hard links. This is related to the next topic about offsets to data.

Offsets to Data

(See Discussion/Proposal)

Some data formats contain header information including a table of offsets in the file to later parts of the data. The ability to directly express offsets within the data (absolute, or relative to some anchor, such as the end of the table of offsets) would make describing these kinds of data files much more direct.

A good example of such a format is TIFF.

Expression Language - Let - Return Construct

In many cases expressions in DFDL's expression language become massively redundant - things that should be expressed once have to be copied and pasted repeatedly (See IPSrcGrp in pcap.dfdl.xsd for an example.)

To fix this, we should enable at least one construct from XQuery to be used in addition to just the XPath-like language specified in DFDL v1.0, and that's the LET-RETURN construct.

A trivial example:

```
dfdl:outputValueCalc='{
    let $x = ../a/b
     return fn:concat($x, $x)
}'
```

It is simply basic software engineering that large and complex things need to be named and reused, not duplicated. DFDL as a language should have features to allow good basic practices to be followed.

Complex Type Derivation - Allowing Properties on ComplexType Definitions

```
<xs:element name="foo" type="myComplexType" dfdl:ref="complexTypeStuff"/>
```

Complex type elements often want to have different default properties than simple types. E.g., dfdl:lengthKind might be explicit for simple types, but implicit for complex types. This makes putting a mixture of simple and complex type elements in the same file painful. If you have a type that you reuse many times, every time you reference the type, you have to add some properties (like dfdl:lengthKind="implicit").

This would be ok, except that you cannot move these added properties down onto the complex type definition. You can with simple types, as the properties on the simple type are combined with those of the element. But complex types don't offer this in DFDL v1.0. This makes you want to use element references a lot in order to hide this noise, but those require global element names, which interferes with namespace management and the very desirable elementFormDefault='unqualified'.

For DFDL 2.0 we should fix these annoyances. We should allow complex type derivations, combining properties on them in the exact same manner as we do for simple types. We should allow properties to be annotated on a complex type definition and for those to be combined with those on an element referencing that type.

Separate LengthKind for Simple and Complex Types

Very often one wants dfdl:lengthKind='delimited' or dfdl:lengthKind='explicit' for simple types, but dfdl:lengthKind="implicit" for complex types. Separating the dfdl:lengthKind into two properties, or having the ability to specify either way, would simplify many schemas that otherwise have a error-prone need to have a dfdl:ref='complex' format reference on every element of complex type to override the default dfdl:lengthKind. That or you have to split the schema and put all simple types in one file (and use only those simple types), and all complex types in another.

Table and Range Lookups / Symbolic Enumerations

(This is now implemented.)

Often one has a representation containing enumerations - integer values - which have symbolic meanings. The parsed result from such data wants to contain strings so the logical infoset is readable and understandable. A means is needed to specify a table of integer constants and their corresponding strings, to be used for parsing, and unparsing. Ranges are a generalization where a symbolic string is used to name all the integers that fall in a range.

Alternative Syntax - Not XML Schema

There is a great deal of resistance to use of XML Schema as the basis for DFDL. It has a big learning curve, and many people avoid it in favor of perceived lighter-weight approches such as JSON.

There are some approaches to XML-Schema that do not express XML Schema itself as XML, but provide a more compact syntax. XSCS (XML Schema Compact Syntax) is one such proposal, which has been investigated. There is no open-source code for this that can be used as a starting point, but the concepts may be adaptable. XSCS does not contain XML Schema's approach to standard annotations, however, so there is work to do here. Other open-source projects such as Apache Spark, have a "struct" notation which may be adaptable.

This approach may also allow one to avoid a number of XML-schema-isms that Daffodil/DFDL is subjected to, such as the PUA mapping problem for XML-disallowed characters.