DFDL plumbing

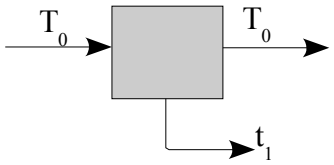
This document is a straw man proposal to try and detail some of the concepts broadly agreed at the DFDL F2F in December 2004.

1 Operational elements

DFDL contains the following operational elements characterised by the types that they take:

- Readers – takes a token off the stream and shortens the stream
- Writers – puts a token on a stream, lengthening it
- Filters – takes a stream and returns a new stream after some manipulation
- Functions – takes one or more simple types and returns a simple type

Reader



The reader is primarily used to populate fields of the logical data. A reader consumes a stream (of a particular type T_0) and produces tokens (usually of a different type t). The stream is modified by the reading such that the token is said to be "consumed" and the next time a reader is applied to the stream an adjacent token will be read. Readers are work forwards through a file.

A simple example of a Reader would take four bytes from an input stream and turn it into an xs:int value, such a Reader might use a parameter like "byteOrder".

Declaration

The declaration of a reader defines:

1. reader's name
2. type of token produced
3. type of token stream connsumed
4. a description (XSD) of all the parameter values that the reader can access

Binding

The basic (Blackbox) definition of a reader, binds the declaration to an external implementation. The details of this binding will be language specific e.g. a Java binding would need to supply:

- fully qualified Java class: com.foo.dfdl.simpleTypes.intFromBytes
- Explicit binding of all the reader's parameters to fields in the class

Use

Readers are used to populate the fields of the logical model. Each element of simple type on a fully annotated DFDL Schema must have annotations specifying:

- which reader is to be used to access this value
- which source stream is to be used to read the value from
- values for all the parameters used by this reader

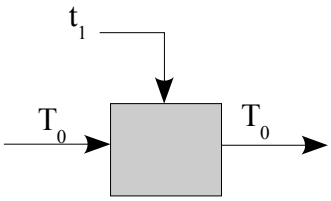
Note that in general most of these values can be set at a default level and the leaves will not need to contain all these details.

Proposal: The reader to use should be specified by setting a parameter value. That way the defaulting of the value can use the same mechanism as the other parameters and does not need a special one.

Proposal: There should be a set of predefined parameter values which correspond to the different xsd simple types: e.g. intReader, floatReader, stringReader etc. These can be set and defaulted at the appropriate level. The reader chosen to populate an element in a logical model is therefor a function of this set of parameters and the type of the field.

Proposal: There should be parameters "source" and "target" which specify the input and output streams in the current context. These will default to the base input and output streams but can be set to other streams.

Writer



The Writer is symmetric with the Reader, it is used to take values from a logical model write them to a target stream. Note that this stream may be used as the input for a reader of another layer of the model (in a multi layered model).

An example of a simple writer is one which would take an xs:int and write 4 bytes corresponding to its value to a stream of bytes, such a writer might use a parameter like "ByteOrder".

Declaration

The declaration of a Writer defines:

1. Writer's name
2. type of token written
3. type of token stream written to
4. a description (XSD) of all the parameter values that the Writer can access

Binding

The basic (Blackbox) definition of a Writer, binds the declaration to an external implementation. The details of this binding will be language specific e.g. a Java binding would need to supply:

- fully qualified Java class: com.foo.dfdl.simpleTypes.intToBytes
- Explicit binding of all the Writers parameters to fields in the class

Use

Writers are used to serialize the logical model. Each element of simple type on a fully annotated DFDL Schema must have annotations specifying:

- which Writer is to be used to access this value
- which target stream is to be used to write the value to
- values for all the parameters used by this Writer

Note that in general most of these values can be set at a default level and the leaf annotations will not need to contain all these details.

Proposal: The Writer should be specified in an analogous way to the Reader in the proposals above.

Filter

The filter is used to carry out manipulations on a stream of data. It produces a new stream of values which may (or may not) be of a different type from the input.

For example a filter might take an stream of characters in, and produce a new stream of characters which was the same as the input but with comments removed.

Declaration

The declaration of a Fliter defines:

1. Filter's name
2. type of token stream consumed
3. type of token stream written out
4. a description (XSD) of all the parameter values that the Filter can access

Binding

The basic (Blackbox) definition of a Filter, binds the declaration to an external implementation. The details of this binding will be language specific e.g. a Java binding would need to supply:

- fully qualified Java class: com.foo.dfdl.simpleTypes.intToBytes
- Explicit binding of all the Filter parameters to fields in the class

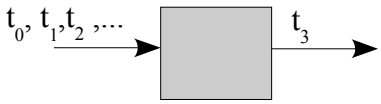
Use

Filters are used to define new stream sources used to populate the logical model (via Readers). A new stream source can be declared in any annotation, although it can only be used within the scope of the bindings of that annotation. The definition of a new source requires:

- the name of the new stream
- which Filter is to be used to produce the new stream
- which target stream is to be used to write the value to
- values for all the parameters used by this Writer

Note that in general most of these values can be set at a default level and the leaf annotations will not need to contain all these details.

Function



A function takes in zero or more values (potentially of different type) and calculates a single value. the values of all the types must be xsd simple types

Question: XPath functions can use values that are XML node trees, do we need to allow this? Can we avoid it? Having this is necessary to e.g. count the number of elements in a sequence, or sum the values of a variable length sequence of elements...

Declaration

The declaration of a Function defines:

1. Functions name
2. type of values input
3. type of value produced

Note: Functions are defined here not to use the parameter context to calculate their values, it parameter values are needed they can be explicitly passed as inputs.

Binding

The basic (Blackbox) definition of a Function, binds the declaration to an external implementation. The details of this binding will be language specific e.g. a Java binding would need to supply:

- fully qualified Java class: com.foo.dfdl.simpleTypes.intToBytes
- The name of the method to call (the method prototype must match the prototype of the new function).

Use

Functions are used in two ways:

1. To set the value of a parameter
2. To set the value of an element in the logical model (derived data)

Proposal: we have used functions to calculate the value for "runtimeOccurs" in past examples. I propose that runtimeOccurs is just a parameter (perhaps one that cannot be overridden or inhereted)

2 Derived operational elements

Reader

New readers can be derived in the following ways:

1. Whitebox definition of a reader
2. Reader derived from a filter

Whitebox

The Whitebox definition of a reader is really a way of encapsulating an intermediate logical data layer for reuse. The definition would lay out a logical XSD/DFDL model in the usual way and a particular

element would be identified as the output element. Reading from this reader would conceptually involve populating this logical model and then accessing the value. Reading a second time would repeat the process.

The new definition must be bound to a Reader declaration with matching prototype.

From Filter

A new reader can be defined using the "ReaderFromFilter" constructor. This takes a filter which produces a stream of *X* and returns a reader which manipulates the input in the same way but can supply one *X* at a time.

The new definition must be bound to a Reader declaration with matching prototype.

Writer

Symetrically to readers, new Writers can be derived in the following ways:

1. Whitebox definition of a Writer
2. Writer derived from a filter

Whitebox

The Whitebox definition would lay out a logical XSD/DFDL model in the usual way these would be populated from an inputstream (since this is part of the output process). Typically this process would be used for adding derived formatting information to the data so many of the fields e.g. explicit length value, would be populated by calculating function values. Once the values were calculated the outputs would be written to a designated target stream.

Note: I'm not sure this works – if we have a white box reader I guess we need (and can build) a white box writer.

The new definition must be bound to a Writer declaration with matching prototype.

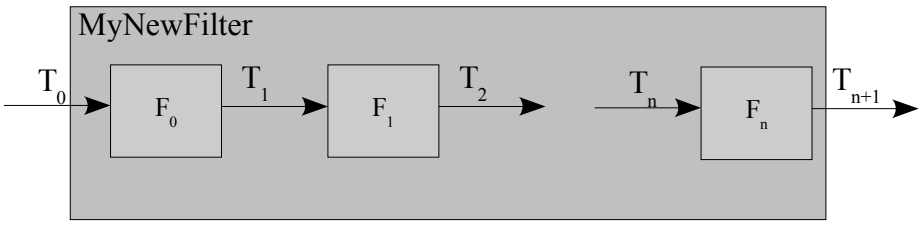
From Filter

A new reader can be defined using the "WriterFromFilter" constructor. This takes a filter which takes a stream of values of type *X* and returns a Writer which manipulates the input data in the same way but can be supplied one *X* at a time.

The new definition must be bound to a Writer declaration with matching prototype.

Filter

Filters can be derived by plumbing together existing Filters. Defining a new Filter allows such plumbing to be encapsulated for reuse. Diagramatically this looks like:



It is tempting to suggest that a filter could be constructed by pluging a writer into a reader. However I thing that this is not the case, the output of this combination would only be a single character, we would have to use something like Mike's recursion to get a many valued stream out. However the same filter could be built by first constructing a filter from the Reader and a filter from the Writer and composing them.

Function

We could declare new functions simply as expressions of existing ones. Do we need this?

3 Scoping of parameters

DFDL will include a large number of parameters. The parameters will define the *context* in which the readers and writers will read and write values. This context has to define, at any simple type in the schema:

1. Which reader is to be used
2. Which writer is to be used
3. Value assignements for all the parameters that those readers and writers can access.
4. The source and target streams that the reader and writer should use.

The parameter context is to be defined by setting parameters in appinfo attributes placed on the XSD model.

The number and complexity of the parameters is such that it is a requirement that we have some mechanism for defaulting the parameter values. It has been agreed that there can be defaults set in the standard, and defaults set at the top level of the document. It has also been agreed that these defaults can be overwritten on the annotation of elements and nodes with XSD simple types (where values will actually be written and read).

It is desirable to be able to scope annotation bindings on branches of the XML tree, but it has been observed that this can raise significant usability issues. The difficulty is that the XML Schema tree can be composed of multiple subtrees in complicated (and recursive ways), there is also a hierarchy of type derivations as well as the hierarchy of elements and attributes. Because of this complexity arbitrary placing of the annotation values can make it difficult to decide at any point in the XML Schema tree what the values of to attributes will be. Further a single point in the Schema tree can correspond to multiple places in the instance tree, it would be possible to end up with situations in which the values of the parameters were different at these different instance sites. i.e. at a given point in the Schema tree there could be multiple values being used for a single parameter!

The problem essentially is that if we scope based on the tree that the XML Schema is defining, this does not correspond to the syntactic scope of the XML Schema document.

Proposal: Scope the parameter bindings should follow the *syntactic scope* of the XML Schema document (see below).

In other words the algorithm for determining the value of a parameter at any point in the tree would be to work your way up the XML of the XML Schema, the lowest definition of that parameter that you find in that traversal is the value that you take. For example consider the following Schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <!-- Global settings -->
  <xs:annotation><xs:appinfo>
    <byteOrder>bigEndian</byteOrder>
  </xs:appinfo></xs:annotation>

  <!-- a complex type xType -->
  <xs:complexType name="xType">
    <xs:sequence>
      <xs:element name="x" type="xs:int"/><!-- byteOrder used here -->
    </xs:sequence>
  </xs:complexType>

  <!-- root element that uses xType -->
  <xs:element name="Root">
    <xs:annotation><xs:appinfo>
      <byteOrder>littleEndian</byteOrder>
    </xs:appinfo></xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="xType" type="xValue"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

In this example the parameter byteOrder is defined twice, once in the global annotation (to be bigEndian) and ones at the top of the Root element (to be littleEndian). The parameter is used in one place to read/write the integer “x” defined within the complex type “xType”.

In this scoping scheme we look at the point of use then look for a definition by ascending the syntactic XML document tree. Here there are no annotations on the complex type xType so the first definition that we encounter is the global one and byteOrder is considered to be bigEndian.

Notice that the annotation on Root does not override this definition because although it is above the “x” element in the instance tree, the xs:element tag to which this annotation is attached does not syntactically surround the xs:element tag in xType where the value is used.

So, this scoping may have some slightly counterintuitive aspects, however it should be clear to human beings, and they can structure their schema to take advantage of the sort of scoping they require.

Parameter properties

Proposal: In addition to the scoping individual parameters can be declared with the following modifiers/properties

- scoped/not scoped – a parameter that is not scoped must be default, or defined at the point of use. An example might be “runtimeOccurs”
- dynamic/static – a dynamic parameter can be set by a runtime value a static one cannot. E.g. intReader must be statically defined.

Parameter validation

It is very desirable to be able to validate that the leaf parameters (those in annotations where xs:simpleTypes are to be read/written) contain only those parameters relevant to the reader and writer

at that leaf. This is difficult however because the assignment of reader and writer will depend on the value of other parameters. If the parameters upon which it depends are all statically defined and a simple scoping mechnism such as the one proposed above is used. It may be possible to define Schematron assertions that will check to ensure that all parameters that are applied are used.