DFDL Virtual XML Prototype

Kristoffer H. Rose, March 3, 2005

Abstract. This is an overview of the Data Format Description Language (DFDL, pronounced “daffodil”) prototype implementation developed to test out how DFDL can be combined with “Virtual XML” to allow access to non-XML data using XML tools yet without requiring complete conversion of the non-XML data into XML. We explain the goals of the project in some detail before we summarize the variant of the DFDL draft specifications that we have implemented, with some issues that are problematic in the specification, as well as a summary of the main implementation challenges.

Table of Contents

1.Introduction

DFDL, the “Data Format Description Language,” is being developed by the Global Grid Forum (GGF) Data Format Description Language Working Group [DFDL-WG]. The primary purpose is to allow reading non-XML data into an XML data structure [XML] but it is also anticipated to be used for writing XML data into non-XML data structures. This is achieved by specifying the desired XML structure with an XML Schema [Schema0] augmented with the details of how the binary data is to be mapped into XML using XML Schema *annotations*. The prototype is based first of all on the DFDL working group's “primer” [DFDL-Primer].

1.1.Goals

The goals of the prototype are:

1. To implement as much of the DFDL specification [DFDL-WG] as possible to learn where the current drafts are underspecified.
2. Allow the use of XPath [XPath2] to access the generated XML instance *without accessing non-essential parts of the data*.
3. Read *business data* such as data X12/EDI [X12], and business programming language "memory dumps" such as C "struct" and COBOL "copybook" structures.

The prototype is available from the "xq-stan" project [XQ] of the IBM "Open Source Bazaar" (further details in the manual section below).

1.2.Status

This document is currently a *working draft* that tracks the design and development of the prototype [XQ]. New features that are still under development are highlighted in pink.

1.3.Terms

The following terms will be used freely in the following.

- **Data.** The file (or other data source) that contains the actual data that is being read by the DFDL interpreter to construct an XML data model instance.
- **DFDL description.** Denotes an XML Schema with DFDL annotations as described in this document.
- **"dfdl" and "xs" namespaces.** Throughout this document we associate the DFDL draft's namespace URI "http://dfdl.org/schema/2004" with the "dfdl:" prefix and, as usual, the XML Schema namespace URI "http://www.w3.org/2001/XMLSchema" is associated with the "xs:" prefix.
- **XML Instance.** Denotes the XML data "document" that is being constructed by the DFDL interpreter from the data. More precisely, the DFDL interpreter constructs an instance of the *XPath/XQuery Data Model* [DM].
- **XML Schema.** An XML document with a description of the *structure* of XML instances.
- **XML Schema particle.** Any of the XML Schema declarations that allow *repetition* (through the minOccurs and maxOccurs "facet" declarations).

1.4.References

DFDL-WG: Martin Westhead, Alan Chappell, and Mike Beckerle, Data Format Description Language Working Group, 2004
XML: Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, Extensible Markup Language (XML) 1.0, 2000, http://www.w3.org/TR/REC-xml
Schema0: David C. Fallside and Priscilla Walmsley, XML Schema Part 0: Primer, 2004
DFDL-Primer: Mike Beckerle, Martin Westhead, GGF DFDL Primer, 2004
XPath2: Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon, XML Path Language (XPath) 2.0, 2004, http://www.w3.org/TR/xpath20/

X12: Data Interchange Standards Association, ASC X12 Standard, , http://www.x12.org
XQ: Kristoffer Rose, Lionel Villard, and Achille Fokoue, XML Query Standard Utilities, 2004, https://w3.opensource.ibm.com/projects/xq-stan/
DM: Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh, XQuery 1.0 and XPath 2.0 Data Model, 2004, http://www.w3.org/TR/xpath-datamodel/
DFDL-Prop: Michael Beckerle, DFDL Representation Properties, 2004, https://forge.gridforum.org/projects/dfdl-wg/document/ggf-dfdl-rep-properties-proposal-002.pdf/en/1

1.5.Plan of the Paper

DFDL is still an evolving standard so in Section 2 we summarize "$DFDL_0$" with the DFDL features that this prototype implements, with some explanation of the differences with the (not yet stable) DFDL draft. In Section 3 we give some examples of $DFDL_0$ specifications. In Section 4 we discuss our implementation and how it generates XML directly from the raw data in an *on-demand* fashion.

2.$DFDL_0$

"$DFDL_0$" is the variant subset of DFDL that our prototype implements. For simplicity, however, we will just use the term "DFDL" in the following and refer to "the DFDL draft" when discussing differences with the GGF DFDL working drafts [DFDL-WG].

In this section we describe the annotations that makes an XML Schema into a DFDL description of how non-XML (binary and textual) data is mapped into XML elements, attributes, and simple values.

Below we will first introduce the directives, then the properties that can be set, detail the low-level mappings associated with the built-in XML Schema atomic types, and finally present the "prolog" of predefined declarations.

2.1.Directives

The DFDL application obtains it's information about how to read the non-XML data from "directives" in "application information" annotations in the XML Schema, like this

```
<xs:annotation><xs:appinfo>
 Directives
</xs:appinfo></xs:annotation>
```

Directives are *not* scoped by default: the directives in an annotation to an XML Schema component apply just to that component. Scoping can be introduced using the definitions directive detailed below where we also precisely state (last) which directives apply to each XML Schema component.

2.1.1.Property definitions

Any number of properties can be bound with the directive

```
<dfdl:properties Property="Value" ... />
```

DFDL properties guide the DFDL mapping from non-XML to XML.

> [**Remark.** This syntax is IMO preferable to the DFDL draft's "property groups" because it will allow easier integration with XPath later by allowing direct access to properties through variables (*i.e.*, $dfdl:byteSize).]

To allow *computation* of properties, the property values are actually *value templates* where the curly brace characters "{" and "}" are special:

- *Single* braces are interpreted as surrounding a *string XPath expression* which will be evaluated for each node generation to obtain the value for that node. Single braces should be matched.
- *Double* braces should be used to insert literal braces. Double braces do not need to be matched.

Be aware that since computations can depend on other computations it is possible to create infinite loops in the XPath computation, similar to "recalculate" loops in spreadsheets.

In addition to the computation, "escapes" are allowed to insert non-XML characters into a (non-computed) property value:

- "\#D...;" ("D..." denotes digits) inserts a single characterswith that code,
- "\#xH...;" ("H..." denotes hexadecimal digits) inserts a single character with that code, and
- "\\#" inserts a literal "\#" character pair.

Note that this is in addition to the use of XML numeric character entities of the form "&#D...;" because XML character entities can only represent the characters that are explicitly allowed by the XML standard [XML].

2.1.2. Simple type use

Integration with simple XML Schema types is declared with a type use declaration

```
<dfdl:use type="Type"/>
```

which requires that there is an XML Schema declaration of the form

```
<xs:simpleType name="Type">
 <xs:restriction base="PrimitiveType"/>
 <xs:annotation><xs:appinfo>
  Directives
 </xs:appinfo></xs:annotation>
</xs:simpleType>
```

The type use declaration indicates to DFDL that each use of the *PrimitiveType* type should be interpreted as a use of the *Type*; in particular it should use the *Directives* as if it they were specified directly for the *PrimitiveType*.

Note that this uses the XML Schema scoping rules to lookup the *Type* but that the "use" directive itself is subject to the DFDL scoping rules.

2.1.3. Configurations

A "configuration" is a collection of related declarations with a name declared with

```
<dfdl:configuration name="ConfigurationName">
  Directives
</dfdl:configuration>
```

where the *Directives* can then be inserted as a group with the other variant of the use directive:

```
<dfdl:use configuration="ConfigurationName"/>
```

Note that a configuration is only visible to use directives where the configuration directive is active, *i.e.*, directly on the component with the annotation containing the configuration directive. This means that configurations and their use must be in annotations to the same XML Schema component unless the configuration is used in a definition directive described below.

2.1.4. Definitions

A "definition section" declares a set of directives by

```
<dfdl:definitions>
  Directives
</dfdl:definitions>
```

where *Directives* should be individual DFDL directives.

Definitions are the mechanism for introducing *scoped* directives: a directive in a definition is propagated to all schema elements that are contained (lexically) in the element that has the definition section annotation.

Specifically we say that a directive is "in scope" for a particular XML Schema declaration if it *either*

- is contained in an annotation directly on the particular XML Schema declaration, *or*
- is contained inside a DFDL definition section in an annotation of
  - ➔ the particular XML Schema declaration,
  - ➔ any other XML Schema declaration that contains the particular one, or
  - ➔ the DFDL "prolog."

Furthermore, property bindings in definitions "shadow" each other so if ever two definitions of the same property, type, configuration, or type use, are both in scope then the inner one (nested within the outer one(s)) is preferred and any directive is preferred over a directive in the prolog.

[**Remark**. This is quite different from the scoping used in the DFDL drafts (so far) where

all properties are implicitly scoped. **T**hose rules are flawed in one important way, IMO: *everything is inherited to all contained components*. This means that it is not possible to define properties for complex types that contain local types inside! The workaround may be to define all types at the top level but this, of course, then makes scoping useless.]

2.1.5.Scoping & context rules

To summarize, here are the rules for finding the value of a property for a given XML Schema component.

1. Search directives in annotations of the schema component itself.
2. Search the "definitions" directives in the lexical context of the schema component (from innermost to outermost).
3. If the component is an element or attribute declared to be of a specific named type then repeat step 1-3 for the schema component of the type declaration.

Searching a "directive" means

1. If it is a "properties" directive then TODO.
2. If it is a "use" directive then TODO

2.2.Properties

DFDL uses properties to determine what (if any) data should be read to create instances of element, attributes, and simple values, from the annotated XML Schema. This section details the supported properties (extracted from [DFDL-Prop] with some simplification).

2.2.1.Data properties

These are properties that describe the low-level characteristics of how the data for a particular value is stored.

Property	*Values*	*Description*
representationType	"text" or "binary"	*Constant*. How the data is stored (default: "text").
byteOffset	integer (≥0)	Offset of the first byte with data (default: 0), relative to the beginning of the *containing* element
byteSize	integer (≥0)	Number of bytes occuppied by the entire data (including initiator, etc.; no default).
byteOrder	"bigEndian" or "littleEndian"	*Constant.* Representation of multi-byte values (default: "bigEndian").
characterSet	string	*Constant.* IANA character set name (for text, default: "UTF-8"), or the special constant "bytes."

2.2.2.Group properties

Properties that describe how a sequence of data is grouped into “units” where this is appropriate, such as between element children or list members. The specified string values are interpreted as character sequences with text and as byte sequences with binary.

Property	*Values*	*Description*
initiator	string	Required before first unit (default: none).
separator	string	Required between units (default: none).
terminator	string	Required (or allowed) after all units (default: none).
finalTerminatorCanBeMissing	boolean	*Constant.* Whether the terminator can be omitted if redundant (default: false).
length	integer (≥0)	The precise occurrence count of units.

2.2.3.Matching properties

Properties that constrain the format of values. Useful both with text (subject to the characterSet property) as well as for binary data (where the regular expression will then match bytes).

Property	*Values*	*Description*
pattern	string	*Constant.* Regular expression (following XML Schema) for matching the textual representation of the value (default: pattern constraining facet if any).
patternGroup	integer	*Constant.* The “group” of the regular expression that actually contains the value, or 0 (the default) to specify that the regular expression matches the entire value.

Regular expressions are explained in detail in the XML Schema standard [Schema0].

[**Remark.** These are not present in the DFDL drafts.]

2.2.4.Numeric text properties

Detail how numerals are represented in text form.

Property	*Values*	*Description*
numberBase	integer	*Constant.* Number base (defaults to 10).

2.2.5.Conditional expressions

A special property is meant to be used in XML schema choice branches to allow disambiguation.

Property	*Values*	*Description*
guard	boolean	Whether to allow the XML Schema “xs:choice” it annotates.

This is mostly useful, of course, in connection with the computed value feature.

2.3.Primitive XML Schema types

We summarize the handling of built-in XML Schema types.

2.3.1.Atomic binary types

The following XML Schema types are active when the “representationType” property is set to “binary.”

Type	*Size*	*Properties*	*Remarks*
xs:byte	1		2-complement signed
xs:short	2	byteOrder	2-complement signed
xs:int	4	byteOrder	2-complement signed
xs:long	8	byteOrder	2-complement signedS
xs:float	4	byteOrder	IEEE format
xs:double	8	byteOrder	IEEE format

2.3.2.Simple text types

The following XML Schema types are active when the “representationType” property is set to “text.”

Type	*Properties*	*Remarks*
xs:string	characterSet, byteOrder, pattern	
xs:long	characterSet, byteOrder, numberBase, isZeroWhenBlank	
xs:integer	characterSet, byteOrder, isZeroWhenBlank	
xs:decimal	characterSet, byteOrder, isZeroWhenBlank	
xs:float	characterSet, byteOrder, isZeroWhenBlank	
xs:double	characterSet, byteOrder, isZeroWhenBlank	

2.3.3.Enumerations

XML Schema enumerations should be encodeable with numeric or binary tags...

2.3.4.Complex types

Complex types define the structure of the XML document generated from the data. Since DFDL uses the schema to *generate* documents special care must be taken to make sure the DFDL engine can determine deterministically what to generate. This is achieved by several measures:

1. There can be only one top-level element declaration.
2. Mixed content is not supported.
3. Every choice must be *guarded* in such a way that at most one choice is applicable for any concrete data. (The choice can either be through XML Schema constraining facets or DFDL directives that restrict the possible data that can generate each choice.)

Other than that there are no restrictions on the use of complex XML Schema constructions.

2.3.5.Restrictions and constraining facets

The following XML Schema “constraining facets” are interpreted by the DFDL prototype.

Facet	*Remarks*
numericBase	
length	
pattern	regular expression

Notice that these constitute constraints on the *generated* values so they are only relevant for the DFDL interpretation of textual data.

2.4.Prolog

The “prolog” is the standard setup for the DFDL engine.

```
<?xml version="1.0" encoding="UTF-8"?><!--*-xml-*-->
<xs:schema
  targetNamespace = "http://dfdl.org/schema/2004"
  elementFormDefault = "qualified" attributeFormDefault = "unqualified"
  xmlns:xs = "http://www.w3.org/2001/XMLSchema"
  xmlns:dfdl = "http://dfdl.org/schema/2004">
```

```
<xs:annotation>
 <xs:documentation>
  $Id: dfdl-prolog.xsd,v 1.8 2005/02/07 15:29:27 krisrose Exp $
  XQ DFDL prototype "prolog schema" with "built-in" types.
 </xs:documentation>

 <!-- Configurations & defaults. -->

 <xs:appinfo>
  <dfdl:definitions>

   <dfdl:configuration name="binaryTypes">
    <dfdl:use type="dfdl:binaryByte"/>
    <dfdl:use type="dfdl:binaryShort"/>
    <dfdl:use type="dfdl:binaryInt"/>
    <dfdl:use type="dfdl:binaryLong"/>
    <dfdl:use type="dfdl:binaryFloat"/>
    <dfdl:use type="dfdl:binaryDouble"/>
    <dfdl:use type="dfdl:binaryByteString"/>
   </dfdl:configuration>

   <dfdl:configuration name="textTypes">
    <dfdl:use type="dfdl:textInteger"/>
    <dfdl:use type="dfdl:textDecimal"/>
    <dfdl:use type="dfdl:textDouble"/>
    <dfdl:use type="dfdl:textFloat"/>
    <dfdl:use type="dfdl:textString"/>
   </dfdl:configuration>

  </dfdl:definitions>
 </xs:appinfo>
</xs:annotation>

<!-- Standard binary types. -->

<xs:simpleType name="dfdl:binaryByte">
 <xs:restriction base="xs:byte"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="1" signed="true"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryShort">
 <xs:restriction base="xs:short"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="2" signed="true"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>
```

```
<xs:simpleType name="dfdl:binaryInt">
 <xs:restriction base="xs:int"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="4" signed="true"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryLong">
 <xs:restriction base="xs:long"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="8" signed="true"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryFloat">
 <xs:restriction base="xs:float"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="4"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryDouble">
 <xs:restriction base="xs:double"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary" byteSize="8"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:binaryByteString">
 <xs:restriction base="xs:string"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="binary"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<!-- Standard text types. -->

<xs:simpleType name="dfdl:textLong">
 <xs:restriction base="xs:long"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="text"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>

<xs:simpleType name="dfdl:textInteger">
 <xs:restriction base="xs:integer"/>
 <xs:annotation><xs:appinfo>
  <dfdl:properties representationType="text" pattern="[-+]?\d+"/>
 </xs:appinfo></xs:annotation>
</xs:simpleType>
```

```
 <xs:simpleType name="dfdl:textDecimal">
  <xs:restriction base="xs:decimal"/>
  <xs:annotation><xs:appinfo>
   <dfdl:properties representationType="text" pattern="[-+]?\d+(?:[.]\d+)?"/>
  </xs:appinfo></xs:annotation>
 </xs:simpleType>

 <xs:simpleType name="dfdl:textFloat">
  <xs:restriction base="xs:float"/>
  <xs:annotation><xs:appinfo>
   <dfdl:properties representationType="text" pattern="[-+]?\d+(?:[.]\d+)?(?:
[Ee][-+]?\d+)?"/>
  </xs:appinfo></xs:annotation>
 </xs:simpleType>

 <xs:simpleType name="dfdl:textDouble">
  <xs:restriction base="xs:double"/>
  <xs:annotation><xs:appinfo>
   <dfdl:properties representationType="text" pattern="[-+]?\d+(?:[.]\d+)?(?:
[Ee][-+]?\d+)?"/>
  </xs:appinfo></xs:annotation>
 </xs:simpleType>

 <xs:simpleType name="dfdl:textString">
  <xs:restriction base="xs:string"/>
  <xs:annotation><xs:appinfo>
   <dfdl:properties representationType="text"/>
  </xs:appinfo></xs:annotation>
 </xs:simpleType>

</xs:schema>
```

3.Examples

Some examples with explanations interspersed. In each case the DFDL annotations are *italicised* for clarity.

3.1.Little-endian binary numbers

The first example justs reads some binary numbers.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--*-xml-*-->
<xs:schema
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dfdl="http://dfdl.org/schema/2004">
```

```
  <!-- DFDL Setup -->
  <xs:annotation>
   <xs:documentation>
    $Id: sequenceOfBinaryLittle.xsd,v 1.8 2005/02/07 15:29:27 krisrose Exp $
    Read various numbers in six fixed binary LITTLE-endian formats.
   </xs:documentation>
   <xs:appinfo>
    <dfdl:definitions>
     <dfdl:use configuration="binaryTypes"/>
     <dfdl:properties byteOrder="littleEndian"/>
    </dfdl:definitions>
   </xs:appinfo>
  </xs:annotation>

  <xs:element name="sextet">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="group" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="byte" type="xs:byte"/>
        <xs:element name="short" type="xs:short"/>
        <xs:element name="int" type="xs:int"/>
        <xs:element name="long" type="xs:long"/>
        <xs:element name="float" type="xs:float"/>
        <xs:element name="double" type="xs:double"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>

 </xs:schema>
```

This will, for example, interpret the byte sequence

```
1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 80 3f 0 0 0 0 0 0 f0 3f
2 2 0 2 0 0 0 2 0 0 0 0 0 0 0 0 0 0 40 0 0 0 0 0 0 0 0 40
```

as the XML

```
<sextet>
 <group>
  <byte>1</byte>
  <short>1</short>
  <int>1</int>
  <long>1</long>
  <float>1.0</float>
  <double>1.0</double>
 </group>
 <group>
  <byte>2</byte>
  <short>2</short>
  <int>2</int>
  <long>2</long>
  <float>2.0</float>
  <double>2.0</double>
 </group>
</sextet>
```

(except without any whitespace).

3.2. Simple COBOL “copybook”

The next example uses constraints to encode the COBOL copybook

```
00572 ******************************************************************
00572 *  COBOL COPYBOOK - CUSTOMERS
00572 *  DATA FOR CUSTOMER TABLE
00572 ******************************************************************
00572  01 CUSTOMER-RECORD.
00573     05 CUSTOMER-LAST-NAME               PIC X(20).
00574     05 CUSTOMER-FILE-FIRST-NAME         PIC X(15).
00575     05 CUSTOMER-FILE-AGE                PIC 999.
00576     05 CUSTOMER-FILE-PHONE              PIC 9(10).
```

in DFDL. Here it is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dfdl="http://dfdl.org/schema/2004">

 <xs:annotation>
  <xs:documentation>
   $Id: copybook.xsd,v 1.4 2005/02/07 15:29:27 krisrose Exp $
   DFDL specification for COBOL COPYBOOK.
  </xs:documentation>
```

```
      <xs:appinfo>
       <dfdl:definitions>
        <dfdl:use configuration="textTypes"/>
        <dfdl:properties characterSet="bytes"/>
       </dfdl:definitions>
      </xs:appinfo>
     </xs:annotation>

     <xs:element name="copybooks">
      <xs:complexType>
       <xs:sequence minOccurs='0' maxOccurs='unbounded'>
        <xs:element name="CUSTOMER-RECORD" >
         <xs:complexType>
          <xs:sequence>

           <xs:element name="CUSTOMER-LAST-NAME" type="xs:string">
            <xs:annotation><xs:appinfo>
              <dfdl:properties byteSize='20' pattern='.{{20}}'/>
            </xs:appinfo></xs:annotation>
           </xs:element>

           <xs:element name="CUSTOMER-FILE-FIRST-NAME" type="xs:string">
            <xs:annotation><xs:appinfo>
              <dfdl:properties byteSize='15' pattern='.{{15}}'/>
            </xs:appinfo></xs:annotation>
           </xs:element>

           <xs:element name="CUSTOMER-FILE-AGE" type="xs:integer">
            <xs:annotation><xs:appinfo>
              <dfdl:properties byteSize="3" pattern='\d{{3}}'/>
            </xs:appinfo></xs:annotation>
           </xs:element>

           <xs:element name="CUSTOMER-FILE-PHONE" type="xs:string">
            <xs:annotation><xs:appinfo>
              <dfdl:properties byteSize="10" pattern='\d{{10}}'/>
            </xs:appinfo></xs:annotation>
           </xs:element>

          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:schema>
```

This will convert the following data (without the quotes and in a single block)

```
"ROSE                KRISTOFFER     0392025555555         "
"ROSE                SOFUS          0060000000000         "
```

to

```
<copybooks>
 <CUSTOMER-RECORD>
  <CUSTOMER-LAST-NAME>ROSE                </CUSTOMER-LAST-NAME>
  <CUSTOMER-FILE-FIRST-NAME>KRISTOFFER     </CUSTOMER-FILE-FIRST-NAME>
  <CUSTOMER-FILE-AGE>39</CUSTOMER-FILE-AGE>
  <CUSTOMER-FILE-PHONE>2025555555</CUSTOMER-FILE-PHONE>
 </CUSTOMER-RECORD>
 <CUSTOMER-RECORD>
  <CUSTOMER-LAST-NAME>ROSE                </CUSTOMER-LAST-NAME>
  <CUSTOMER-FILE-FIRST-NAME>SOFUS          </CUSTOMER-FILE-FIRST-NAME>
  <CUSTOMER-FILE-AGE>6</CUSTOMER-FILE-AGE>
  <CUSTOMER-FILE-PHONE>0000000000</CUSTOMER-FILE-PHONE>
 </CUSTOMER-RECORD>
</copybooks>
```

3.3.Prefix-tagged text

This simple example illustrates how XML Schema “choice” is allowed as long as the choices are made mutually exclusive based on the binary data.

```
<?xml version="1.0" encoding="UTF-8"?><!--*-xml-*-->
<xs:schema
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dfdl="http://dfdl.org/schema/2004">

 <xs:annotation>
  <xs:documentation>
   $Id: sequenceOfTextInitiated.xsd,v 1.6 2005/02/07 15:29:27 krisrose Exp $
   Read some textual numbers with a prefix letter indicating the format.
  </xs:documentation>

  <xs:appinfo>
   <dfdl:definitions>
    <dfdl:use configuration="textTypes"/>
   </dfdl:definitions>
  </xs:appinfo>
 </xs:annotation>

 <xs:element name="list">
  <xs:complexType>
   <xs:choice minOccurs="1" maxOccurs="unbounded">

    <xs:element name="integer" type="xs:integer">
     <xs:annotation><xs:appinfo>
       <dfdl:properties initiator="I" terminator=" "
          finalTerminatorCanBeMissing="1"/>
     </xs:appinfo></xs:annotation>
    </xs:element>
```

```
      <xs:element name="decimal" type="xs:decimal">
       <xs:annotation><xs:appinfo>
         <dfdl:properties initiator="D" terminator=" "
            finalTerminatorCanBeMissing="1"/>
       </xs:appinfo></xs:annotation>
      </xs:element>

      <xs:element name="floating" type="xs:double">
       <xs:annotation><xs:appinfo>
         <dfdl:properties initiator="F" terminator=" "
            finalTerminatorCanBeMissing="1"/>
       </xs:appinfo></xs:annotation>
      </xs:element>

     </xs:choice>
    </xs:complexType>
   </xs:element>

  </xs:schema>
```

Running this on a simple text such as

```
"I1 F1 I2 D2.0 "
```

(without the quotes) will generate the output XML

```
<list>
 <integer>1</integer>
 <floating>1.0</floating>
 <integer>2</integer>
 <decimal>2.0</decimal>
</list>
```

(again without any actual white space in the output).

3.4.“Geek code” (X12)

Our next example is a sample of X12/EDI [X12] borrowed from the GNU XML::Edifact project samples. We assume the following “UN/ECE” dictionary:

```
CED    COMPUTER ENVIRONMENT DETAILS
Function: To give a precise definition of all necessary elements
          belonging to the configuration of a computer system
          like hardware, firmware, operating system,
          communication (VANS, network type, protocol, format)
          and application software.
```

```
010   1501  COMPUTER ENVIRONMENT DETAILS QUALIFIER         M  an..3
020   C079  COMPUTER ENVIRONMENT IDENTIFICATION            M
      1511   Computer environment, coded                   C  an..3
      1131   Code list qualifier                           C  an..3
      3055   Code list responsible agency, coded           C  an..3
      1510   Computer environment                          C  an..35
      1056   Version                                       C  an..9
      1058   Release                                       C  an..9
      7402   Identity number                               C  an..35
```

This is encoded using regular expression annotations as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  elementFormDefault="qualified" attributeFormDefault="unqualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:dfdl="http://dfdl.org/schema/2004">

 <xs:annotation>
  <xs:documentation>
   $Id: geek.xsd,v 1.8 2005/02/07 15:29:27 krisrose Exp $
   Read X12/EDI-encoded Geek code.
  </xs:documentation>
  <xs:appinfo>
   <dfdl:definitions>
    <dfdl:use configuration="textTypes"/>
    <dfdl:properties characterSet="ASCII"/>
   </dfdl:definitions>
  </xs:appinfo>
 </xs:annotation>

 <!-- Root element -->
 <xs:element name="COMPUTER-ENVIRONMENT-DETAILS">
  <xs:annotation><xs:appinfo>
    <dfdl:properties initiator="CED"/>
  </xs:appinfo></xs:annotation>

  <xs:complexType>
   <xs:sequence>

    <xs:element name="COMPUTER-ENVIRONMENT-DETAILS-QUALIFIER"
         type="xs:integer">
     <xs:annotation><xs:appinfo>
       <dfdl:properties initiator="+"/>
     </xs:appinfo></xs:annotation>
    </xs:element>

    <xs:element name="COMPUTER-ENVIRONMENT-IDENTIFICATION">
     <xs:complexType>
      <xs:sequence>
```

```
          <xs:element name="Computer-environment-coded" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="[+]([^:+]*)" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Code-list-qualifier" type="xs:string">
           <xs:annotation><xs:appinfo>
            <dfdl:properties pattern="(?:[:]([^:+]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Code-list-responsible-agency-coded" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="(?:[:]([^:+]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Computer-environment" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="(?:[:]([^:+]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Version" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="(?:[:]([^:+]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Release" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="(?:[:]([^:]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

          <xs:element name="Identity-number" type="xs:string">
           <xs:annotation><xs:appinfo>
             <dfdl:properties pattern="(?:[:]([^:]*))?" patternGroup="1"/>
           </xs:appinfo></xs:annotation>
          </xs:element>

        </xs:sequence>
       </xs:complexType>
      </xs:element>

     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:schema>
```

The above will, for example, convert the text

```
CED+2+:::Linux:1.2:13
```

to the XML document

```
<COMPUTER-ENVIRONMENT-DETAILS>
  <COMPUTER-ENVIRONMENT-DETAILS-QUALIFIER>2</COMPUTER-ENVIRONMENT-DETAILS-QUALIFIER>
  <COMPUTER-ENVIRONMENT-IDENTIFICATION>
    <Computer-environment-coded></Computer-environment-coded>
    <Code-list-qualifier></Code-list-qualifier>
    <Code-list-responsible-agency-coded></Code-list-responsible-agency-coded>
    <Computer-environment>Linux</Computer-environment>
    <Version>1.2</Version>
    <Release>13</Release>
    <Identity-number></Identity-number>
  </COMPUTER-ENVIRONMENT-IDENTIFICATION>
</COMPUTER-ENVIRONMENT-DETAILS>
```

(without indentations or line feeds).

4.Implementation

Here we describe the implementation.

4.1.Manual

We explain how to run and the srandard examples.

4.1.1.Unpack

Unpack the “dfdlmilestoneX.zip” in an otherwise empty directory:

```
$ unzip dfdlmilestoneX.zip
```

This will create a bunch of files, among them the “dfdl-prolog.xsd” and executable “rundfdl.jar”.

4.1.2.Run

I'll assume that you have a Java 1.4 executable as the command “java” and have set the current directory to the directory where unzip was run above.

Running

```
$ java -jar rundfdl.jar dfdl.xsd data [XPath]
```

will interpret the “*dfdl*.xsd” XML Schema file as a DFDL script and use it to read the “*data*” file. The generated XML is then output; if there is an “XPath” argument then only the subtrees of the nodes selected by the XPath are output.

Here is a brief description of the files; the “README” file also has some interesting XPath expressions to add.

Example	*DFDL schema*	*Sample data*
Little-endian binary numbers	sequenceOfBinaryLittle.xsd	onetwolittle.data
COBOL copybook	copybook.xsd	copybook.text
Prefix-tagged text numbers	sequenceOfTextInitiated.xsd	onetwoinitiated.text
EDI-style “geek code”	geek.xsd	geek.text

4.2.Design

The code is part of the “xq-stan” IIOSB project [XQ] and follows the conventions of that project.

4.2.1.Generic data model

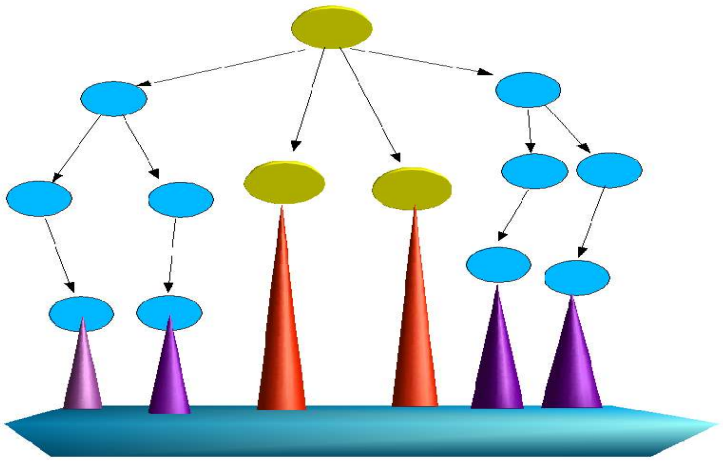
Specifically,

- The XML that is generated can be accessed as a “SequenceFocus” light-weight data model instance.
- It uses (and can be accessed through) the xq-stan XQuery (and XPath/XSLT) components.
- It can be used with SAX/DOM.

4.2.2.Virtual XML

A few guiding principles from “Virtual XML” were used.

- The XML is generated in a *lazy* fashion: we implement the XML Data Model [DM]in such a way that XML nodes are generated and thus no data is read from the data until nodes are actually *accessed*.
- The internal form of the materialized tree is not exposed thus it lends itself to compaction.



Specifically the implementation completely separates

- what needs to be read from the data to determine the *size in bytes* of the data that maps to any given subtree, and
- what needs to be read to actually populate the tree.

For example, when using the XPath

4.3.Specific limitations

Here are some specific limitations & issues.

- XML Schema “references” are not resolved. *Should be easy to fix.*
- I have generalized the “mappedTypeSet” notion of the DFDL drafts to “configuration” to allow other declarations than “use.”
- The repetition properties (length, separator) do not work yet. *Should be implemented together with segment compaction.*
- It is not presently possible to generate attributes. *Since attributes from DFDL's point of view are just like elements this is just a day of careful duplication once the element code is finished.*
- The “numberBase” property only works for target types that are restrictions of xs:long, and base can only be up to 36. *Not easy to fix as it is inherited from Java.*
- DFDL has lots of properties for numeric formatting that should be incorporated as this is quite easy (but time consuming), there are also the data formats. *It will be several weeks of work to properly support all the variations, locales, etc., even when leveraging the Java basic implementation.*
- The XPath implementation is not complete but quite useable. *The XPath implemenation is generic for the whole virtual XML effort. The status is summarized in the eval/doc/status.html file of the project [XQ].*