Experiments in Data Format Interoperation Using Defuddle

A Technical Note prepared as part of the National Archives and Records Administration (NARA) funded "Innovative Systems and Software: Applications to NARA Research Problems" project.

Robert E. McGrath, Jason Kastner, Jim Myers
Cyberenvironments and Technologies Directorate
National Center for Supercomputing Applications
University of Illinois, Urbana-Champaign

September 2009

Abstract

This document discusses the status of the Defuddle parser and recent work conducted as part of the “Innovative Systems and Software: Applications to NARA Research Problems” project.

Robust sharing, reuse, and curation of data requires a clean separation of issues related to bits, formats, and logical content. To address these issues the Open Grid Forum is defining the Data Format Description Language (DFDL) standard for describing the structure of binary and textual files and data streams so that their format, structure, and metadata can be exposed as XML [3]. While this is sufficient for describing the internal layout of data (the “syntax”), interoperability and curation also require description of logical relationships within and between data sets in terms of globally understood concepts (the “semantics”). Extending the concept of the DFDL, and the Defuddle DFDL parser implementation, we have defined a two-step declarative mechanism for describing the structure and relations in binary and ASCII data in terms of vocabularies defined using standard Semantic Web languages (RDF and OWL). We briefly outline our approach below and highlight the potential benefits of exposing internal data semantics in the context of larger semantic systems.

DFDL and Defuddle separate the issues:

- The DFDL annotate schema defines the logical structure of the data
- The DFDL annotations map the bits to the logical structure
- The XML output of Defuddle provides a standard format with a well-defined logical structure
- The RDF output of Defuddle provides description of logical relationships in a standard format

This document describes two demonstrations of data interoperation with Defuddle. These demonstrations explore how Defuddle and DFDL can be used for tasks that usually are implemented with code. The first demonstration explores an aspect of file characterization, namely recognition of the “MIME-type” of a file. The second demonstration illustrates a simple example of recognition and 3D data format interoperation. In both these cases, the Defuddle parser software was used without modification. Each application required development of appropriate DFDL-annotated XML schemas to read the binary or text data. Also, the MIME-type recognizer used the semantic extensions to produce an RDF triple that asserts the MIME-type.

The rest of the paper is organized as follows. Section 2 gives some background on the Defuddle technology. Section 3 explains the file identification demonstration. Section 4 discusses the 3D formats demonstration. Section 5 concludes. Appendices give details of the XML schemas and XSL transformst used.

Contents

1. Introduction

Robust sharing, reuse, and curation of data requires a clean separation of issues related to bits, formats, and logical content. To address these issues the Open Grid Forum is defining the Data Format Description Language (DFDL) standard for describing the structure of binary and textual files and data streams so that their format, structure, and metadata can be exposed as XML [3]. While this is sufficient for describing the internal layout of data (the "syntax"), interoperability and curation also require description of logical relationships within and between data sets in terms of globally understood concepts (the "semantics"). Extending the concept of the DFDL, and the Defuddle DFDL parser implementation, we have defined a two-step declarative mechanism for describing the structure and relations in binary and ASCII data in terms of vocabularies defined using standard Semantic Web languages (RDF and OWL). We briefly outline our approach below and highlight the potential benefits of exposing internal data semantics in the context of larger semantic systems.

DFDL and Defuddle separate the issues:

- The DFDL annotate schema defines the logical structure of the data
- The DFDL annotations map the bits to the logical structure
- The XML output of Defuddle provides a standard format with a well-defined logical structure
- The RDF output of Defuddle provides description of logical relationships in a standard format

This document describes two demonstrations of data interoperation with Defuddle. These demonstrations explore how Defuddle and DFDL can be used for tasks that usually are implemented with code. The first demonstration explores an aspect of file characterization, namely recognition of the "MIME-type" of a file. The second demonstration illustrates a simple example of recognition and 3D data format interoperation. In both these cases, the Defuddle parser software was used without modification. Each application required development of appropriate DFDL-annotated XML schemas to read the binary or text data. Also, the MIME-type recognizer used the semantic extensions to produce an RDF triple that asserts the MIME-type.

The rest of the paper is organized as follows. Section 2 gives some background on the Defuddle technology. Section 3 explains the file identification demonstration. Section 4 discusses the 3D formats demonstration. Section 5 concludes. Appendices give details of the XML schemas and XSL transformst used.

2. Background

The Data Format Description Language (DFDL) is a draft standard specification from the Open Grid Forum (http://forge.gridforum.org/projects/dfdl-wg).[1] DFDL proposes to describe existing data formats, both binary and text, in a manner that makes the data accessible through generic

[1] The DFDL specification is still under development and has not released an official draft. This discussion in this section is based on an early draft of the specification which includes features that have since been removed from consideration as part of a version 1.0 specification but are expected to be re-introduced in later iterations. The semantic extensions discussed here can be applied with minor modifications to later versions of the DFDL or to other implementations of the DFDL.

mechanisms. The DFDL specification is based on XML Schema, a standard that defines the structure and semantics of XML document formats. In analogy with an XML parser, which can use an XML schema to interpret XML input in terms of the logical model in the schema, a DFDL parser can interpret an input that is a sequence of bytes (ASCII or binary, not necessarily XML) in terms of schema. In both cases, the output is an XML Information Model (for DFDL, *as if* the input was XML).

XML Schema allows annotation of schemas, a feature primarily used for the benefit of human readers, but also usable by applications. DFDL uses this mechanism to add information regarding how to apply the formal structure of an XML schema to arbitrary data file formats. DFDL annotations specify low-level format issues such as whether the source format is ASCII or binary and, for binary, whether "big-endian" or "little-endian" encodings have been used. They also specify higher level associations between the raw bytes on the disk and the logical data model specified by the XML schema including which bytes are associated with which XML elements and how to interpret bytes in a particular format as control structures.

The open source Defuddle parser [8, 17] was originally developed within the DOE-funded Scientific Annotation Middleware (SAM) project (www.scidac.org/SAM/) and has been further developed at NCSA [8, 17] as both a proof of concept of the DFDL specification and a mechanism for testing concepts which can feed back into the specification process. A specific aim of Defuddle is to demonstrate that an efficient, generic DFDL parser can be built and that such a parser can effectively address real-world examples.

Defuddle extends the concepts embedded in the Java XML Binding (JAXB) Specification). Specifically, Defuddle extends the open source Apache JaxMe (http://ws.apache.org/jaxme/) implementation of JAXB and follows the same 2-step model of parsing the schema (XML Schema or DFDL format description for JaxMe and Defuddle respectively) to generate Java source code for the classes required to parse and represent data in the relevant format followed by a step in which those classes are compiled and run on instances of that format. In both cases, the result is an in-memory representation of the data that is logically equivalent to the model expressed in the schema and which can be manipulated directly within an application or exported into a new XML file. Thus data can be manipulated according to its logical structure (specified by the XML schema), regardless of how it is physically represented.

2.1. Semantic Extensions to Defuddle

While the XML Schema language is well suited for describing the layout of data (the "syntax"), interoperability and curation also require description of logical relationship within and between data in terms of globally understood concepts (the "semantics").

The DFDL model converts data into XML; or, alternatively, extracts XML from the data; in order to gain the advantages of the XML standard for data representation. Our goal in this task has been to extend this model to extract descriptions of the structure and relations in the data into standard semantic web languages (the Resource Description Framework (RDF) (http://www.w3.org/TR/rdf-concepts/) and the Web Ontology Language (OWL) (http://www.w3.org/TR/owl-features/).

We extend the JaxME/Defuddle approach, to extract descriptions of the structure and relations in the data into standard semantic web languages (the Resource Description Framework (RDF)

(http://www.w3.org/TR/rdf-concepts/) and the Web Ontology Language (OWL) (http://www.w3.org/TR/owl-features/). Our approach is a two-step process, the standard DFDL processing within Defuddle to generate XML, and a second phase to extract semantic descriptions from the XML (see Figure 3). The second phase uses GRDDL as a standard mechanism for *declaring these transformations*.

The RDF output from the semantically enhanced Defuddle can be considered both as a file and as a collection of triples. (When more than one transformation is applied, all the triples are aggregated into a single collection.) The initial implementation generates XML-encoded RDF and simply exports that as a file. However, the Defuddle-derived triples should be used as part of a larger graph of information including provenance, domain ontologies, and social network information. To support this, the RDF triples would need to be ingested into some form of triple store where they can be queried and used in subsequent logical inference, rule-based, and other processes. Given NCSA's extensive work in developing semantic content middleware (Tupelo [5, 6], http://www.tupeloproject.org), it would be natural to store Defuddle's output via this middleware. Preliminary work in this direction has begun as part of a Google Summer Of Code Student Fellowship in which Yigang Zhou will be implementing a mechanism (as a Tupelo Context) to invoke Defuddle as part of a file upload operation in Tupelo and to store the output as metadata or new content linked to the original data via provenance information (see http://socghop.appspot.com/student_project/show/google/gsoc2009/ncsa/t124022775909).

3. MIME Type Recognition (File Identification)

For data interoperation and curation, it is often critical to characterize a data object (e.g., a file) in order to determine what further processing should be applied. The first step of this process is to identify the (presumptive) format of a digital object, from which it may be possible to validate the object, extract data and metadata, and/or transform the object to another format.

Identification is usually done by recognizing signatures of well-known types of objects, especially, objects that conform to widely used standards. A "signature" could be any logical property of the stored bits, but it is most often deliberately inserted as part of a standard, e.g., as a required pattern in the first few bytes of the stored file. The identification process consists of testing the digital object against a set of rules, until one or more matches to indicate the putative "type" of the data object.

Projects such as JHOVE2 [1], ExifTool, [7], Metadata Extraction tool [14], ffident [15], and the File Information Tool Set (FITS) [4] provide mechanisms to identify the (presumptive) format of a digital object.[2] JHOVE2 uses DROID and PRONOM, which define a (custom) simple language for describing signatures in data objects [2]. Essentially, JHOVE2 has a collection of known signatures, against which a candidate file is compared. Other utilities use similar approaches, though there is no universal standard for defining the signatures. FITS wraps JHOVE and several other similar services [4].

In other work as part of "Innovative Systems and Software: Applications to NARA Research Problems" project, three-dimensional file formats are identified by programs that encapsulate the

[2] In addition to identification, many of these have facilities to validate the object as well as identifying it.

signature of the files [9-13]. This approach is similar to JHOVE2, FITS, and the other works cited above, though not based on the same technology. It is important to note that file identification is only a minor goal of McHenry et al.. They have developed techniques for multistep translations of three dimensional data files, along with automatic assessments of the quality of such conversions.

3.1. Example Implementation

We observe here that file identification can also be addressed using DFDL. A set of signatures are declared in a DFDL-annotated XML schema, which produces a simple XML output to indicate the purported type of the file. The GRDDL phase generates an RDF triple to assert the putative type of the object.

An example implementation illustrates this approach. In this approach, the Defuddle parser is given a special DFDL-annotated schema, which encapsulates a collection of logical signatures. The output is a simple XML file with a tag to indicate the discovered type.

First, a DFDL schema must be created to "peek" at the initial bytes of an input file. This data is read into an array of bytes, represented in XML. This data can be read into XML elements, where they can be examined. Appendix A shows the schema with example annotations.

Ideally, this data should be in a temporary, "hidden layer", which is not written to the output XML. We note that, to date, this layer concept is not supported in Version 1 of the DFDL standard, though it may be included in later versions. The Defuddle parser implements a form of layers, though the current implementation was not adequate for this example.

After the data elements, the DFDL schema has a "choice" element which selects on the basis of matches in the data. If the signature matches, the XML output will have an element that represents the "hit". E.g., if the first few bytes are "GIF89a", then the output will have a <gif> tag. The matches are done via the DFDL `<condition>` tag.[3] The conditions encode the decimal numbers of the signature.

The Defuddle parser generates code to parse the input files, and the result is an XML file with a tag for one of the known formats, or else no tag. Figure 1 shows example output when the input file is a (binary) GIF file.

```
<?xml version='1.0' encoding='UTF-8'?>
<dfdl:SIGNATURE xmlns:ns0="http://www.w3.org/2003/g/data-view#"
ns0:transformation=".../examples/schemas/MIMEDetect.xsl " xmlns:dfdl="DFDL">
[…]

  <dfdl:gif>-42</dfdl:gif>
</dfdl:SIGNATURE>
```

Figure 1. Example output.

Of course, there is little purpose for this XML file itself. What we really want is a logical assertion that "This file has MIME type 'image/gif'" or whatever, which can be used in

[3] Defuddle implements an early version of this feature, Version 1 of the DFDL standard will define a full specification.

automated reasoning (e.g., to select processing modules or other schemas to apply). This can be achieved through the semantic extensions to Defuddle.

An XSL stylesheet is created that matches the XML tags for each recognized type, and generates an RDF triple to assert the MIME type. GRDDL is used to apply this transformation to the output of the first phase, such as Figure 1. Appendix B shows an example XSL style sheet.

The output is an RDF triple that asserts the type or else is empty if there is no match. Figure 2 shows and example of this triple (as written in RDF XML binding).

```
<rdf:Description rdf:about="<inputfile">
  <hasMIMEtype>
        "image/gif"
      <hasMIMEtype>
</rdf:Description>
```

Figure 2. Example RDF output.

3.1. Discussion

This section has illustrated the use of Defuddle as a file identifier. While there are many simple ways to achieve this goal, this use case illustrates some of the advantages of DFDL/Defuddle for preservation and archiving.

There is one schema that selects among all the known formats. (Adding a new format requires adding a new choice into the schema.) The recognition rules are encoded explicitly in the schema, not in executable code. The schema is a standard language "executed" by any conforming parser. Similarly, the RDF extraction is done through explicit rules encoded in standard XSL.

This example illustrates the use of two features of Defuddle. The 'conditions' are used to mimic the XML 'choice' operation, based on data in the file. This, in turn, works best with 'hidden layers', which read data into addressable structures, but are not written as part of the output model. In this example, the layer is important because we need to read the beginning of the file as bytes, and then examine it several times looking for matching signatures. Only when there is a match, can we "consume" the data and produce the intended output.

The examples show in this report implement simple pattern matches. However, the Defuddle schema could detect more complex signatures, with conditional logic or possible nested choices.

In this application, the Defuddle parser only reads the significant part of the file, such as the first 100 bytes. For this reason, the performance will not depend on the size or complexity of the file. However, the signatures are applied in order, so the parser could potentially become slow if the number of signatures becomes very large. If this became an issue, the DFDL schema could be divided into more than one schema, which could be applied in parallel.

4. Data Interoperation for 3D Formats

Three dimensional objects are used in many applications including CAD, GIS, games, and computer graphics. Three dimensional objects (real or fictional) can be represented as complex graphs, labeled with attributes. Furthermore, even for a given representation there are many ways

it might be serialized for storage or transfer. Since there is no body of accepted practice, a plethora of "3D file formats" have developed and become widely used.[4] This diversity presents a significant pragmatic problem for software that needs to interoperate with data from many sources.

Consider a three dimensional object represented in two different formats. Software that needs to use objects encoded in both formats must align the relevant information with the concepts used in the program. The alternative data represents the same concepts, the spatial layout of a 3D object. If two formats both describe 3D objects, so they are, in principle, logically comparable and interoperable (i.e., usable in a given program). Of course, not every data item can be compared or translated; in some cases some data can be represented in one but not the other format.

One approach is to import each object into common data structures using a custom "reader". This is commonly used in graphics software. The reader encapsulates a conceptual mapping of the linearized object to the desired data structure, including translations and transformations, and possibly creating default place holders for "missing" elements. It is up to the creator of the reader software to assure that the semantics are correctly translated—the elements in the file must be correctly interpreted, and data structures output must be correct and correctly represent the input. Since this semantic mapping is encoded in software, it is not always easy to discern what is implemented and whether it is correct. When using multiple formats, there must be a reader for each format, and the chain is only as strong as its weakest link—if even one of the readers is incorrect (or missing), the results will suffer.

In other work as part of "Innovative Systems and Software: Applications to NARA Research Problems" project, McHenry et al. have developed techniques for multistep translations of three dimensional data files, along with automatic assessments of the quality of such conversions [9-13]. This system wraps a variety of existing "readers" and "writers", along with intelligence about their characteristics. To transform a file from format A to format B, a sequence of transformations is discovered, from A to M, M to N, etc. and eventually to B. When more than one path exists, the optimal path may be determined, depending on constraints on the results and capabilities of the transformations.

Defuddle provides an alternative approach, in which the conceptual mappings are described in an open schema, rather than embedded in code. The data format is described by an XML schema annotated to specify the data elements in the file. Defuddle reads the data file and generates an XML information model that conforms to the schema, populated by the (non-XML) data from the file. This XML can be further processed by standard XML-aware tools, or written out to an XML file or database.

Usually, Defuddle will extract the data into XML that represents the logical layout of the input file. This is done because it is natural for the Defuddle schema to define XML elements representing the elements in the data file. However, this means that the Defuddle schema for two different formats will be different, and the resulting XML will not match. Fortunately, XML is easily processed and transformed to other XML, so the two data formats can be used together.

[4] For purposes of this note, a "3D file format" is considered to be a linearization of 3D data structures, i.e., a two way mapping between a memory representation of 3D data and a linear sequence of bits. This might be a specification for storage layout, and it might be defined through software (e.g., a library). The linearization might include a variety of additional data, and may be part of a general "data format".

One XML model can be translated to another, through XSL stylesheets to describe the transformations to be applied. As in the case of the DFDL schema, the XSL represents the conceptual mapping in an open, declarative language.

Several types of transformation can be imagined, and they are not mutually exclusive. In some cases, it may be possible and useful to transform XML representing one format to XML representing another. So, a file of type 1 would be extracted to XML schema 1. A file of type 2 would be read into XML schema 2. Then the XML matching schema 2 would be transformed with XSL into an instance of schema 1 (or vice versa). In this way, data from file 1 and file 2 could be used together.

Another approach is to map the extracted XML into a third XML schema that represents the comparable elements. Data from each file would be transformed into XML representations, then each would be transformed with XSL into a third XML schema.

4.1. Experiments with Example 3D Formats

To illustrate this concept, we developed an example DFDL schema to read examples of 3D data into XML. The examples are:

- The ISO STEP standard [16].
- The OBJ format [18].

In this exploration, it was immediately clear that developing a detailed DFDL schema for something as complex as a 3D file format is a daunting challenge. We created rather simple and elementary DFDL schemas that gloss over many of the details of the input files. Even so, the resulting XML, crude as it may be, is potentially quite useful because it can be processed with standard tools to reformat or extract the data of interest.

STEP Data

The STEP format is an ISO standard, designed to exchange for 3D engineering data [16]. We created a DFDL schema for a subset of the standard.

The STEP standard defines hundreds of keywords which represent the geometry, constituents, properties, and metadata that describe arbitrarily complex 3D objects. These may be represented in many forms, including a standard file format called EXPRESS.[5] For instance, the shape of an object is represented by a set of "CARTESIAN_POINT" records, which are organized into a collection of "VERTEX" and "SURFACES". The file may also describe folds and curves, and also may include metadata.

A DFDL description could model this data in many ways. A simple alternative is to model the outermost STEP concepts found in an EXPRESS file. More complex approaches might describe structure implied by the nesting in EXPRESS. Even the simple strategy is extremely useful because the XML can be processed by standard tools to extract or reformat to meet the needs of the using software. Appendix C shows an example schema that uses this strategy.

[5] The STEP Technical Committee has defined an XML binding for STEP (ISO 10303-28:2007). For some purposes, DFDL might seek to create XML which conforms to this standard, but other uses might want to extract data to other schemas.

Table 1 shows examples from our "shallow" description of the STEP file. The left column lists selected items from the EXPRESS file, and the right column shows XML generated by Defuddle.

Table 1. Fragments of STEP and the XML generated by Defuddle

STEP (EXPRESS)	Defuddle output
DATA; #10=DESIGN_CONTEXT('3D Mechanical Parts',#83,'design'); #11=PRODUCT_DEFINITION('A','First version',#53,#10); …	<dfdl:datablock> <dfdl:DataLabelStart>DATA</dfdl:DataLabelStart> <dfdl:Data3D> <dfdl:DataIndex>#10</dfdl:DataIndex> <dfdl:DataFunction>DESIGN_CONTEXT</dfdl:DataFunction> <dfdl:DataValues>'3DMechanicalParts'</dfdl:DataValues> <dfdl:DataValues>#83</dfdl:DataValues> <dfdl:DataValues>'design'</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataIndex>#11</dfdl:DataIndex> <dfdl:DataFunction>PRODUCT_DEFINITION</dfdl:DataFunction> <dfdl:DataValues>'A'</dfdl:DataValues> <dfdl:DataValues>'Firstversion'</dfdl:DataValues> <dfdl:DataValues>#53</dfdl:DataValues> <dfdl:DataValues>#10</dfdl:DataValues> </dfdl:Data3D> …
#100=ADVANCED_BREP_SHAPE_REPRESENTATION('brep_rep',(#88),#99); #101=CLOSED_SHELL('',(#102,#103,#104,#105,#106,#107)); #102=ADVANCED_FACE('',(#108),#164,.T.); …	<dfdl:DataFunction>ADVANCED_BREP_SHAPE_REPRESENTATION</dfdl:DataFunction> <dfdl:Data3D> <dfdl:DataIndex>#101</dfdl:DataIndex> <dfdl:DataFunction>CLOSED_SHELL</dfdl:DataFunction> <dfdl:DataValues>'brep_rep'</dfdl:DataValues> <dfdl:DataValues>(#88)</dfdl:DataValues> <dfdl:DataValues>#99</dfdl:DataValues> </dfdl:Data3D> <dfdl:DataValues>''</dfdl:DataValues> <dfdl:DataValues>(#102</dfdl:DataValues> <dfdl:DataValues>#103</dfdl:DataValues> <dfdl:DataValues>#104</dfdl:DataValues> <dfdl:DataValues>#105</dfdl:DataValues> <dfdl:DataValues>#106</dfdl:DataValues> <dfdl:DataValues>#107)</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataIndex>#102</dfdl:DataIndex> <dfdl:DataFunction>ADVANCED_FACE</dfdl:DataFunction> <dfdl:DataValues>''</dfdl:DataValues> <dfdl:DataValues>(#108)</dfdl:DataValues> <dfdl:DataValues>#164</dfdl:DataValues> <dfdl:DataValues>.T.</dfdl:DataValues> </dfdl:Data3D>

#120=ORIENTED_EDGE(',*,*,#145,.F.); #121=ORIENTED_EDGE(',*,*,#146,.F.); …	<dfdl:DataFunction>ORIENTED_EDGE</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>*</dfdl:DataValues> <dfdl:DataValues>*</dfdl:DataValues> <dfdl:DataValues>#145</dfdl:DataValues> <dfdl:DataValues>.F.</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataIndex>#121</dfdl:DataIndex> <dfdl:DataFunction>ORIENTED_EDGE</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>*</dfdl:DataValues> <dfdl:DataValues>*</dfdl:DataValues> <dfdl:DataValues>#146</dfdl:DataValues> <dfdl:DataValues>.F.</dfdl:DataValues> </dfdl:Data3D>
#156=VERTEX_POINT(',#238); #157=VERTEX_POINT(',#239); …	<dfdl:Data3D> <dfdl:DataIndex>#156</dfdl:DataIndex> <dfdl:DataFunction>VERTEX_POINT</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>#238</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataIndex>#157</dfdl:DataIndex> <dfdl:DataFunction>VERTEX_POINT</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>#239</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D>
#170=CARTESIAN_POINT(',(72.,-3.,21.3646461429269)); #171=CARTESIAN_POINT(',(72.,-3.,18.8646460766809)); …	<dfdl:Data3D> <dfdl:DataIndex>#170</dfdl:DataIndex> <dfdl:DataFunction>CARTESIAN_POINT</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>(72.</dfdl:DataValues> <dfdl:DataValues>-3.</dfdl:DataValues> <dfdl:DataValues>21.3646461429269)</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataIndex>#171</dfdl:DataIndex> <dfdl:DataFunction>CARTESIAN_POINT</dfdl:DataFunction> <dfdl:DataValues>"</dfdl:DataValues> <dfdl:DataValues>(72.</dfdl:DataValues> <dfdl:DataValues>-3.</dfdl:DataValues> <dfdl:DataValues>18.8646460766809)</dfdl:DataValues> </dfdl:Data3D>

Wavefront OBJ

"OBJ" files are a widely used format for exchanging 3D data used in computer graphics and games, among other applications. OBJ is much simpler than STEP, defining relatively few keywords, and representing geometry as a set of vertices and faces (e.g., see [18]).

A DFDL description is straightforward, defining elements that correspond to the keywords in OBJ. Appendix D gives an example schema.

Table 2. Sample elements from OBJ and the XML generated by Defuddle

OBJ	Defuddle Output
# using: ply2obj v1.3 # from: stl2ply v1.1	<?xml version='1.0' encoding='UTF-8'?> <dfdl:Wavefront xmlns="http://vesta.ncsa.uiuc.edu/Temp/SCSV.xsd" xmlns:dfdl="DFDL"> <dfdl:metadatablock> <dfdl:Comment># using: ply2obj v1.3</dfdl:Comment> <dfdl:Comment># from: stl2ply v1.1</dfdl:Comment>
# 18 vertices v 72.000 -8.500 13.864 v 72.000 -8.500 17.614 v 72.000 -8.500 21.364	<dfdl:Comment># 18 vertices</dfdl:Comment> </dfdl:metadatablock> <dfdl:datablock> <dfdl:Data3D> <dfdl:DataType>v</dfdl:DataType> <dfdl:DataValues>72.000</dfdl:DataValues> <dfdl:DataValues>-8.500</dfdl:DataValues> <dfdl:DataValues>13.864</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataType>v</dfdl:DataType> <dfdl:DataValues>72.000</dfdl:DataValues> <dfdl:DataValues>-8.500</dfdl:DataValues> <dfdl:DataValues>17.614</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataType>v</dfdl:DataType> <dfdl:DataValues>72.000</dfdl:DataValues> <dfdl:DataValues>-8.500</dfdl:DataValues> <dfdl:DataValues>21.364</dfdl:DataValues> </dfdl:Data3D>
g headusOBJexport # 32 polygons f 6 7 9 f 17 16 9 f 9 7 10 …	<dfdl:Data3D> <dfdl:DataType>g</dfdl:DataType> <dfdl:DataValues>headusOBJexport</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataType>#</dfdl:DataType> <dfdl:DataValues>32</dfdl:DataValues> <dfdl:DataValues>polygons</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataType>f</dfdl:DataType> <dfdl:DataValues>6</dfdl:DataValues> <dfdl:DataValues>7</dfdl:DataValues> <dfdl:DataValues>9</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D> <dfdl:DataType>f</dfdl:DataType> <dfdl:DataValues>17</dfdl:DataValues> <dfdl:DataValues>16</dfdl:DataValues> <dfdl:DataValues>9</dfdl:DataValues> </dfdl:Data3D> <dfdl:Data3D>

	<dfdl:DataValues>9</dfdl:DataValues> <dfdl:DataValues>7</dfdl:DataValues> <dfdl:DataValues>10</dfdl:DataValues> </dfdl:Data3D>

4.2. Extracting Metadata and Transforming the Data

As illustrated above, the XML generated from each type of 3D format reflects the contents and semantics of that format. But once translated into XML, it is relatively easy to process the XML to extract and format the data for different purposes.

For example, to extract the vertices and polygons from the mesh described in the two object, an XSL style sheet could be used for each case. For the STEP data, the XSL would select from the DataFunction tags, to find the CARTESIAN_POINTS, VERTEX, and so on, which would be written out as vertices. For the OBJ data, the XSL would select from the 'DataType' tags to find the 'v', which would be written as vertices.

As in the example discussed in Section 3, the semantic extensions to Defuddle can be used to extract metadata and create RDF. The OBJ file has very little metadata, but STEP has significant metadata in the header and elsewhere. Figure 4 shows and example XSL style sheet to extract some metadata from the STEP XML produced by Defuddle. The results are show in Figure 5. In this example, Defuddle has extracted each metadata field as a single string, even though some of the fields have further structure.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xsl:stylesheet [
         <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
       <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
         <!ENTITY dfdl 'DFDL'>

         <!-- RDF doesn't allow local URIs -->
         <!ENTITY absdfdl 'http://ncsa.uiuc.edu/DFDL#'>
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;"
                xmlns:dfdl="&dfdl;" xmlns:absdfdl="&absdfdl;"
                xmlns:dc="http://purl.org/dc/elements/1.1/"
                xmlns:stp="http://test.com/stp"
                version="1.0">

  <xsl:output method="xml" version="1.0" encoding="utf-8" indent="yes"/>

  <xsl:template match="/dfdl:STP">
    <rdf:RDF>
      <rdf:Description>
        <rdf:type rdf:resource="&absdfdl;STP"/>
        <stp:filedesc><xsl:value-of
select="dfdl:metadatablock/dfdl:FileDescription"/></stp:filedesc>
```

```
        <stp:filename><xsl:value-of
select="dfdl:metadatablock/dfdl:FileName"/></stp:filename>
        <stp:fileschema><xsl:value-of
select="dfdl:metadatablock/dfdl:FileSchema"/></stp:fileschema>
      </rdf:Description>
    </rdf:RDF>
  </xsl:template>

</xsl:stylesheet>
```

Figure 3. Example XSL transformation to extract basic metadata from STP XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
   xmlns:_10="http://test.com/"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
>
  <rdf:Description rdf:nodeID="psFtwDDR4">

<_10:stpfiledesc>FILE_DESCRIPTION(/*description*/(''),/*implementation_level*
/'2;1'</_10:stpfiledesc>

<_10:stpfileschema>FILE_SCHEMA(('CONFIG_CONTROL_DESIGN')</_10:stpfileschema>

<_10:stpfilename>FILE_NAME(/*name*/'c_airHandlingRoom',/*time_stamp*/'2007-
02-06T10:20:16-
05:00',/*author*/(''),/*organization*/(''),/*preprocessor_version*/'ST-
DEVELOPERv8',/*originating_system*/'',/*authorisation*/''</_10:stpfilename>
    <rdf:type rdf:resource="http://ncsa.uiuc.edu/DFDL#STP"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 4. Example RDF extracted from STP. These results did not filter the STEP 'comments'.

4.3. Comparison with Procedural Translation and other Methods

It is clearly possible to translate data with conversion programs, or via a combination of loaders, transforms, and writers (e.g., see [9-13]). Any method that relies on a conversion program or script captures the conversion mapping (i.e., the critical knowledge) in code.

In comparison, DFDL / Defuddle with XSL explicitly defines the conversion in descriptions of the data and the transformations. Since these descriptions are written in an open standard language, then they can be processed by any conforming software. This is the great advantage for interoperability and longevity: it is necessary to transmit and preserve the description and the specification, but not the software to execute it. Any conforming implementation should get the same result (in theory, anyway).

Of course, there is a cost associated with the declarative approach. Manipulating data through generic parsers may be less efficient than with custom software. The cost depends on details of the data as well as the implementations (e.g., for one example, see [8]).

Possibly more important than run-time performance, creating schemas and DFDL markup is not simple, nor is creating XSL. In many cases, it will be easier and cheaper to write one or a few readers especially when sample code is widely available. One way to consider this, is that there

are costs to develop transformations, and costs to port and preserve the knowledge in the transformations. Using code defers some of the costs to the time when the readers must be ported to new platforms. When the data must be preserved for the future, the readers must be preserved as well—which is difficult to say the least. In comparison, creating a DFDL schema and XSL pays most of these costs immediately, with the hope reaping benefits at later times.

4.4. Semantics

The example above illustrated how the structure of the data can be extracted and transformed. It is equally important to represent and preserve the semantics of these data. For example, the STEP data defines concepts about curves and surfaces. OBJ has no such concepts, and other formats have yet other concepts. Merely extracting the data into XML is not sufficient, it is critical to know what "ADVANCED_FACE", "FACE_OUTER_BOUND", "EDGE_LOOP", and so on mean. Similarly, we assume, but probably should not, that the STEP concept "CARTESIAN_POINT" and "VERTEX" together are equivalent to the OBJ concept "v".

The semantic extensions to Defuddle ([8, 17]) can be used to extract many kinds of relations from the data files. The RDF can, for example, associate terms with standard ontologies. It can also define relations among the data elements within a file (e.g., STEP implicitly links data elements via their index in the file). In the example above, it might be possible to associate the STEP and OBJ concepts with as standard ontology, which would enable automated reasoning to determine if the data are actually comparable or not. The RDF can also associate elements in different files and stores, e.g., to associate the data with software and data that produced it.

5. Conclusion

This note described explorations of two scenarios for data interoperability using Defuddle. The first example uses a single DFDL/Defuddle schema to identify the file "type" by matching a signature. The semantic extension to Defuddle is used to generate RDF to assert the type. This example provides equivalent results to other existing systems, using an open, standard language to describe the patterns.

The second example used Defuddle to transform 3D data into XML. The DFDL schema defines the logical model of the data, which can be used to extract metadata or to further transform the data. One advantage of transforming the data into XML is that it can then be transformed into other logical models, to enable interoperation of data from several formats. Again, these transformations are done through descriptions rather than procedural code.

Acknowledgements

This work was supported through National Science Foundation Cooperative Agreement NSF OCI 05-25308 and Cooperative Support Agreements NSF OCI 04-38712 and NSF OCI 05-04064by the National Archives and Records Administration.

References

1. Abrams, Stephen, Sheila Morrissey, and Tom Cramer, *"What? So What?" The Next-Generation JHOVE2 Architecture for Format-Aware Characterization*, in *iPRES 2008:*

The Fifth International Conference on Preservation of Digital Objects. 2008: London. http://confluence.ucop.edu/download/attachments/3932229/Abrams_a70_pdf.pdf?version=1

2. Brown, Adrian, *Automatic Format Identification Using PRONOM and DROID*. The National Archives of the United Kingdom DPTP-01, London, 2006. http://www.nationalarchives.gov.uk/aboutapps/fileformat/pdf/automatic_format_identification.pdf
3. DFDL-WG, *Data Format Description Language (DFDL)*. 2005. http://forge.gridforum.org/projects/dfdl-wg
4. FITS. *File Information Tool Set (FITS)*. 2009, http://code.google.com/p/fits/.
5. Futrelle, Joe. *Harvesting RDF Triples*. In *IPAW'06 International Provenance and Annotation Workshop* 2006, 64-72.
6. Futrelle, Joe, Jeff Gaynor, Joel Plutchak, James D. Myers, and Robert E. McGrath, *Tupelo: a Framework for E-Science Knowledge Spaces (accepted)* in *Microsoft Escience Workshop*. 2009: Pittsburg.
7. Harvey, Phil. *ExifTool by Phil Harvey*. 2009, http://www.sno.phy.queensu.ca/~phil/exiftool/.
8. McGrath, Robert E., Jason Kastner, Alejandro Rodriguez, and Jim Myers, *Towards a Semantic Preservation System*. National Center for Supercomputing Applications, Urbana, 2009. http://cet.ncsa.uiuc.edu/publications/Semantic_Preservation_System.pdf
9. McHenry, K. and P. Bajcsy, *Key Aspects in 3D File Format Conversions*, in *Joint Annual Meeting of the Society of American Archivists and the Council of State Archivists, 2009 Research Forum "Foundations and Innovations"*. 2009: Austin. http://www.archivists.org/publications/proceedings/researchforum/2009.asp
10. McHenry, Kenton and Peter Bajcsy, *3D Data Analysis*, in *WVU/NETL/ERA Workshop on Digital Preservation of Complex Engineering Data*. 2009: Morgantown, WV.
11. McHenry, Kenton and Peter Bajcsy, *Towards a Universal Converter (submitted)*, in *SPIE 2010*. 2009.
12. McHenry, Kenton, Rob Kooper, and Peter Bajcsy, *Taking Matters into Your Own Hands: Imposing Code Reusability for Universal File Format Conversion (submitted)*, in *Microsoft eScience Workshop*. 2009: Pittsburg.
13. McHenry, Kenton, Rob Kooper, and Peter Bajcsy, *Towards a Universal, Quantifiable, and Scalable File Format Converter*, in *5th International IEEE eScience conference*. 2009: Oxford, UK.
14. National Library of New Zealand. *Metadata Extraction Tool*. 2009, http://meta-extractor.sourceforge.net/.
15. Schmidt, Marco. *ffident— Java metadata extraction / file format identification library*. 2006, http://web.archive.org/web/20061106114156/http://schmidt.devlib.org/ffident/index.html.
16. STEP Tools Incorporated. *ISO 10303 STEP Standards*. 2009, http://www.steptools.com/library/standard/.
17. Talbott, Tara D., Karen L. Schuchardt, Eric G. Stephan, and James D. Myers, *Mapping Physical Formats to Logical Models to Extract Data and metadata: The Defuddle Parsing Engine*, in *International Provenance and Annotation Workshop*. 2006, Springer: Heidelberg. p. 73-81.
18. Wikipedia. *Obj*. 2009, http://en.wikipedia.org/wiki/Obj.

Appendix A: MIMEDetect Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="DFDL"
xmlns:dfdl="DFDL"
xmlns="DFDL">
      <!-- this type would be used in a hidden layer →
     <xs:complexType name="ByteArrayType">
                <xs:sequence>
                    <xs:element name="array" type="xs:byte" maxOccurs="100"/>
                </xs:sequence>
            </xs:complexType>
       <xs:element name="SIGNATURE">
       <xs:complexType name="foo">
                    <xs:sequence>
<!-- a non-hidden array of bytes -->
                    <xs:element name="array" type="xs:byte" maxOccurs="100"/>
<!--  the “hidden layer”
                    <xs:annotation>
                           <xs:appinfo source="http://dataformat.org/">
                               <dfdl:dataFormat>
                                 <dfdl:RepresentationLayer name="array"
type="ByteArrayType" maxOccurs="100"/>
                               </dfdl:dataFormat>
                           </xs:appinfo>
                    </xs:annotation>
-->
                    <xs:choice>
                           <xs:element name="gif" type="xs:byte">
                                  <xs:annotation>
                                         <xs:appinfo>
                                                <dfdl:dataFormat xmlns:dfdl="DFDL">
                                                       <dfdl:charset>US-
ASCII</dfdl:charset>
                                                       <condition>((../array[0] ==
'71')AND(../array[1] == '73')AND(../array[2] == '70')AND(../array[3] ==
'56')AND(../array[4] == '57')AND(../array[5] == '97'))</condition>
<!--
0000000   G   I   F   8   9   a  \f  \0  \f  \0 367  \0  \0 377 377 377
        4947 3846 6139 000c 000c 00f7 ff00 ffff
         71,73,70,56, 57, 97  (or 55)
-->
                                                </dfdl:dataFormat>
                                         </xs:appinfo>
                                  </xs:annotation>
                           </xs:element>
                           <xs:element name="jpg" type="xs:byte" >
                                  <xs:annotation>
                                         <xs:appinfo>
                                                <dfdl:dataFormat xmlns:dfdl="DFDL">
                                                       <dfdl:charset>US-
ASCII</dfdl:charset>
                                                       <condition>((../array[6] ==
'74')AND(../array6] == '70')AND(../array[8] == '73')AND(../array[9] ==
'70'))</condition>
<!--
0000000 377 330 377 340  \0 020   J   F   I   F  \0 001 001 001  \0   `
```

```
        d8ff e0ff 1000 464a 4649 0100 0101 6000
Hex:
skip 6 bytes, 4a,46,49,46 majvers, minvers
             74, 70, 73, 70
-->
                                                    </dfdl:dataFormat>
                                              </xs:appinfo>
                                       </xs:annotation>
                                </xs:element>
                                <xs:element name="stp" type="xs:string" >
                                       <xs:annotation>
                                              <xs:appinfo>
                                                    <dfdl:dataFormat xmlns:dfdl="DFDL">
                                                          <dfdl:charset>US-
ASCII</dfdl:charset>
                                <condition>((../array[0] == '73')AND(../array[1] ==
'83')AND(../array[2] == '79')AND(../array[3] == '45')AND(../array[4] ==
'49')AND(../array[5] == '48')AND(../array[6] == '51')AND(../array6] ==
'48')AND(../array[8] == '51')AND(../array[9] == '45')AND(../array[10] ==
'50')AND(../array[11] == '49')AND(../array[12] == '59'))</condition>
<!--
    ISO-10303-21;
73, 83, 79, 45, 49, 48, 51, 48, 51, 45, 50, 49, 59
-->
                                                    </dfdl:dataFormat>
                                              </xs:appinfo>
                                       </xs:annotation>
                                </xs:element>
                         </xs:choice>
                         </xs:sequence>
                  </xs:complexType>
            </xs:element>
</xs:schema>
```

Appendix B: MIMEDetect XSL

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE xsl:stylesheet [
         <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
        <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
         <!ENTITY dfdl 'DFDL'>

         <!-- RDF doesn't allow local URIs -->
         <!ENTITY absdfdl 'http://ncsa.uiuc.edu/DFDL#'>
]>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;"
                xmlns:dfdl="&dfdl;" xmlns:absdfdl="&absdfdl;"
                xmlns:dc="http://purl.org/dc/elements/1.1/"
                version="1.0">

  <xsl:output method="xml" version="1.0" encoding="utf-8" indent="yes"/>

  <xsl:template match="/">
       <xsl:apply-templates select="/dfdl:SIGNATURE"/>
  </xsl:template>

  <xsl:template match="dfdl:SIGNATURE">
       <xsl:apply-templates select="./dfdl:stp"/>
       <xsl:apply-templates select="./dfdl:jpg"/>
       <xsl:apply-templates select="./dfdl:gif"/>
  </xsl:template>

  <xsl:template match="dfdl:stp">
    <rdf:RDF>
      <rdf:Description rdf:about="URN:theSourceNeededHere">
         <hasMIMEtype>
            "application/step"
         </hasMIMEtype>
      </rdf:Description>
    </rdf:RDF>
  </xsl:template>

  <xsl:template match="dfdl:jpg">
    <rdf:RDF>
      <rdf:Description rdf:about="URN:theSourceNeededHere">
         <hasMIMEtype>
            "image/jpg"
         </hasMIMEtype>
      </rdf:Description>
    </rdf:RDF>
  </xsl:template>

  <xsl:template match="dfdl:gif">
    <rdf:RDF>
      <rdf:Description rdf:about="URN:theSourceNeededHere">
         <hasMIMEtype>
            "image/gif"
         </hasMIMEtype>
      </rdf:Description>
    </rdf:RDF>
  </xsl:template>

</xsl:stylesheet>
```

Appendix C: STEP Schema

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

	<xs:element name="STP" type="STPFile">
	</xs:element>
	<xs:complexType name ="STPFile">
		<xs:sequence>
			<xs:element name="ISOStart" type="ISOLabel"/>
			<xs:element name="metadatablock" type="STPMeta"/>
			<xs:element name="datablock" type="STPData"/>
			<xs:element name="ISOEnd" type="ISOLabel"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="ISOLabel">
		<xs:sequence>
			<xs:element name="ISOLabel" type="xs:string">
				<xs:annotation>
					<xs:appinfo>
						<dfdl:terminator kind="regexp or
string">;</dfdl:terminator>
						<dfdl:charset>US-ASCII</dfdl:charset>
					</xs:appinfo>
				</xs:annotation>
			</xs:element>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="STPMeta">
		<xs:sequence>
			<xs:annotation>
				<xs:appinfo>
					<dfdl:dataFormat>
						<dfdl:repType>text</dfdl:repType>
					</dfdl:dataFormat>
				</xs:appinfo>
			</xs:annotation>
			<xs:element name="HeaderLabel" type="xs:string">
				<xs:annotation>
					<xs:appinfo>
						<dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
						<dfdl:terminator kind="regexp or
string">;</dfdl:terminator>
						<dfdl:charset>US-ASCII</dfdl:charset>
					</xs:appinfo>
				</xs:annotation>
			</xs:element>
			<xs:element name="FileDescription" type="xs:string">
				<xs:annotation>
					<xs:appinfo>
						<dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
						<dfdl:terminator kind="regexp or
string">\\);</dfdl:terminator>
						<dfdl:charset>US-ASCII</dfdl:charset>
					</xs:appinfo>
```

```
                        </xs:annotation>
                  </xs:element>
                  <xs:element name="FileName" type="xs:string">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                    <dfdl:terminator kind="regexp or
string">\\);</dfdl:terminator>
                                    <dfdl:charset>US-ASCII</dfdl:charset>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
                  <xs:element name="FileSchema" type="xs:string">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                    <dfdl:terminator kind="regexp or
string">\\);</dfdl:terminator>
                                    <dfdl:charset>US-ASCII</dfdl:charset>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
                  <xs:element name="HeaderLabelEnd" type="xs:string">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                    <dfdl:terminator kind="regexp or
string">ENDSEC;</dfdl:terminator>
                                    <dfdl:charset>US-ASCII</dfdl:charset>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
            </xs:sequence>
      </xs:complexType>
      <xs:complexType name="STPData">
            <xs:sequence>
                  <xs:annotation>
                        <xs:appinfo>
                              <dfdl:dataFormat>
                                    <dfdl:repType>text</dfdl:repType>
                              </dfdl:dataFormat>
                        </xs:appinfo>
                  </xs:annotation>
                  <xs:element name="DataLabelStart" type="xs:string">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                    <dfdl:terminator kind="regexp or
string">;</dfdl:terminator>
                                    <dfdl:charset>US-ASCII</dfdl:charset>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
                  <xs:element name="Data3D" type="DataType" maxOccurs="unbounded">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:dataFormat>
                                          <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
```

```
                                                <dfdl:terminator kind="regexp or
string">ENDSEC;</dfdl:terminator>
                                                <dfdl:charset>US-ASCII</dfdl:charset>
                                          </dfdl:dataFormat>
                                    </xs:appinfo>
                              </xs:annotation>
                        </xs:element>
                  </xs:sequence>
            </xs:complexType>
            <xs:complexType name="DataType">
                  <xs:sequence>
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:dataFormat>
                                          <dfdl:repType>text</dfdl:repType>
                                    </dfdl:dataFormat>
                              </xs:appinfo>
                        </xs:annotation>
                        <xs:element name="DataIndex" type="xs:string">
                              <xs:annotation>
                                    <xs:appinfo>
                                          <dfdl:dataFormat>
                                                <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                                <dfdl:terminator kind="regexp or
string">=</dfdl:terminator>
                                                <dfdl:charset>US-ASCII</dfdl:charset>
                                          </dfdl:dataFormat>
                                    </xs:appinfo>
                              </xs:annotation>
                        </xs:element>
                        <xs:element name="DataFunction" type="xs:string">
                              <xs:annotation>
                                    <xs:appinfo>
                                          <dfdl:dataFormat>
                                                <dfdl:terminator kind="regexp or
string">\\(</dfdl:terminator>
                                                <dfdl:charset>US-ASCII</dfdl:charset>
                                          </dfdl:dataFormat>
                                    </xs:appinfo>
                              </xs:annotation>
                        </xs:element>
                        <xs:element name="DataValues" type="xs:string"
maxOccurs="unbounded">
                              <xs:annotation>
                                    <xs:appinfo>
                                          <dfdl:dataFormat>
                                                <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                                <dfdl:separator kind="regexp or
string">,</dfdl:separator>
                                                <dfdl:terminator kind="regexp or
string">\\);</dfdl:terminator>
                                                <dfdl:charset>US-ASCII</dfdl:charset>
                                          </dfdl:dataFormat>
                                    </xs:appinfo>
                              </xs:annotation>
                        </xs:element>
                  </xs:sequence>
            </xs:complexType>
</xs:schema>
```

Appendix D: OBJ Schema

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

      <xs:element name="Wavefront" type="WavefrontFile">
      </xs:element>

      <xs:complexType name ="WavefrontFile">
            <xs:sequence>
                  <xs:element name="metadatablock" type="WavefrontMeta"/>
                  <xs:element name="datablock" type="WavefrontData"/>
            </xs:sequence>
      </xs:complexType>

      <xs:complexType name="WavefrontMeta">
            <xs:sequence>
                  <xs:annotation>
                        <xs:appinfo>
                              <dfdl:dataFormat>
                                    <dfdl:repType>text</dfdl:repType>
                              </dfdl:dataFormat>
                        </xs:appinfo>
                  </xs:annotation>
                  <xs:element name="Comment" type="xs:string" maxOccurs="3">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:separator kind="regexp or
string">\\n</dfdl:separator>
                                    <dfdl:charset>US-ASCII</dfdl:charset>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
            </xs:sequence>
      </xs:complexType>

      <xs:complexType name="WavefrontData">
            <xs:sequence>
                  <xs:annotation>
                        <xs:appinfo>
                              <dfdl:dataFormat>
                                    <dfdl:repType>text</dfdl:repType>
                              </dfdl:dataFormat>
                        </xs:appinfo>
                  </xs:annotation>
                  <xs:element name="Data3D" type="DataType"
maxOccurs="unbounded">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:dataFormat>
                                          <dfdl:ignore kind="regexp or
string">\\p{Space}</dfdl:ignore>
                                          <dfdl:charset>US-
```

```
ASCII</dfdl:charset>
                                    </dfdl:dataFormat>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
            </xs:sequence>
      </xs:complexType>

      <xs:complexType name="DataType">
            <xs:sequence>
                  <xs:annotation>
                        <xs:appinfo>
                              <dfdl:dataFormat>
                                    <dfdl:repType>text</dfdl:repType>
                              </dfdl:dataFormat>
                        </xs:appinfo>
                  </xs:annotation>
                  <xs:element name="DataType" type="xs:string">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:dataFormat>
                                          <dfdl:terminator kind="regexp or
string">\\p{Blank}+</dfdl:terminator>
                                          <dfdl:charset>US-
ASCII</dfdl:charset>
                                    </dfdl:dataFormat>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
                  <xs:element name="DataValues" type="xs:string"
maxOccurs="unbounded">
                        <xs:annotation>
                              <xs:appinfo>
                                    <dfdl:dataFormat>
                                          <dfdl:separator kind="regexp or
string">\\p{Blank}+</dfdl:separator>
                                          <dfdl:terminator kind="regexp or
string">\\n|\\r|[\\r\\n]</dfdl:terminator>
                                          <dfdl:charset>US-
ASCII</dfdl:charset>
                                    </dfdl:dataFormat>
                              </xs:appinfo>
                        </xs:annotation>
                  </xs:element>
            </xs:sequence>
      </xs:complexType>
</xs:schema>
```