Data Format Description Language: Lessons Learned
Concepts and Experience

Robert E. McGrath
National Center for Supercomputing Applications
University of Illinois, Urbana-Champaign
September, 2011

This report was prepared as part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, 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.

Many individuals contributed to the work discussed here, including James D. Myers, Tara Gibson, Jason Kastner, Alejandro Rodriguez, and Joe Futrelle.

The opinions expressed are those of the author.

Abstract

For the past 6 years as part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, NCSA has contributed to the development of the Open Grid Forum (OGF) standard format description language, the Data Format Description Language (DFDL). A DFDL parser is sufficient to support interpretation of arbitrary binary or ASCII formatted files in terms of well-defined logical models.

The Data Format Description Language emerged from a variety of unrelated projects and products, which had various goals and approaches. The goal of the OGF DFDL-WG is to build on previous experience to create a consensus standard that can replace the disparate related efforts. In 2011, the DFDL specification was accepted as a "Proposed Recommendation" of the Open Grid Forum.

The DFDL is a critical new technology for many important use cases, including:

- Access and manipulation of non-XML data, such as data from sensors or simulations
- Interoperation of data from many independent sources
- Preservation of access to data for long periods of time
- Construction and access to "virtual datasets" from many sources.

This capability is especially interesting for archives that need to preserve access to data for long periods of time.

Beyond maintaining the accessibility of the raw '1's and '0's of digital data, preservation and interoperation requires maintaining an ability to interpret the data as meaningful structures, relationships, and visual representations. NCSA has investigated concepts for a general descriptive method for accessing data in arbitrary file formats and providing interpreted information of it in XML and RDF representations, supporting discovery and long-term preservation of content.

This technology has broad application across the curation and preservation processes, and more broadly in e-Science in general, and the DFDL has been identified by the US National Archives and Record Administration (NARA) as a priority in the area of Human Computer Interaction and Information Management ([9], p. 12).

This project has included contributions to the development of the DFDL standard, test implementations of the concepts, and explorations of semantic extensions for DFDL. This document summarizes the activities and presents some lessons learned in the course of this project.

1. Introduction

For the past 6 years as part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, we have contributed to the development of the Open Grid Forum standard format description language (the Data Format Description Language (DFDL) [13, 51]) and format-independent parser (in earlier work, Defuddle [30, 57], and later the Daffodil parser [52]). A DFDL parser is sufficient to support interpretation of arbitrary binary or ASCII formatted files in terms of well-defined logical models as well as mechanisms to support integration of data, metadata, and provenance information from different systems to assemble such models.

The Data Format Description Language emerged from a variety of unrelated projects and products, which had various goals and approaches. The goal of the OGF DFDL-WG is to build on previous experience to create a consensus standard that can replace the disparate related efforts. In 2011, Version 1 of the DFDL specification was accepted as a "Proposed Recommendation" of the Open Grid Forum.

A DFDL description is, in principle, a "declarative" program for reading data. Any DFDL-enabled parser can read a DFDL schema and automatically generate software to read the data. This declarative model means that a valid schema is sufficient to access the data, it is not necessary to have specific software.

This is a critical technology for many important use cases, including:

- Access and manipulation of non-XML data, such as data from sensors or simulations
- Interoperation of data from many independent sources
- Preservation of access to data for long periods of time
- Construction and access to "virtual datasets" from many sources.

This capability is especially interesting for archives that need to preserve access to data for long periods of time. Given a DFDL specification for a class of datasets, we can assure access in the future by preserving the DFDL. There is no need to preserve a software stack or reader. DFDL does not provide capabilities that cannot be achieved with procedural or scripted software, but it requires far fewer dependencies to remain viable in the future. Such a framework would dramatically lower the per-file-format effort required for preservation.

Beyond maintaining the accessibility of the raw '1's and '0's of digital data, preservation and interoperation requires maintaining an ability to interpret the data as meaningful structures, relationships, and visual representations. We have investigated concepts for a general descriptive method for accessing data in arbitrary file formats and providing interpreted information of it in XML and RDF representations, supporting discovery and long-term preservation of content.

This technology has broad application across the curation and preservation processes, and more broadly in e-Science in general. DFDL can be used to extract metadata from files

rather than data (descriptive information rather than all the data values) for use in assessment and cataloging. DFDL solves important problems as part of a broader effort to support data curation and preservation through the development of capabilities to automate the extraction and integration of data, metadata, and provenance information from distributed sources. Such a capability is becoming essential as data volumes increase and as researchers embrace "data-intensive" techniques. Management of large ensembles of research results, documenting the connection of results to the input data and algorithms used to create them, tracking use of data products, enabling integration of data in different formats, and preserving the entire corpus of information for future use are increasingly becoming the barriers to scientific progress.

The DFDL has been identified by the US National Archives and Record Administration (NARA) as a priority in the area of Human Computer Interaction and Information Management ([9], p. 12). Pursuant to this goal as part of a continuing collaborative agreement between the National Center for Supercomputing Applications and the National Archives and Record Administration, we have investigated the development of DFDL. This work has included contributions to the development of the DFDL standard, test implementations of the concepts, and explorations of semantic extensions for DFDL. This document summarizes some lessons learned in the course of this project.

Section I establishes the background for the work, defining important concepts, discussing earlier work that led to the standard, and sketching key use cases (and some "anti-use cases"). Section II discusses implementation of the standard, and describes our own prototyping efforts. Section III considers extensions to the DFDL, including semantic extensions and integration with workflows and tools. Finally, Section IV wraps up with future work and a summary.

I. Background: History, Concepts, and Use Cases

2. Overview

The Data Format Description Language is a language for describing data, specifically a data format. In this context, a *data format* is a specified set of rules for representing data in digital form, in files or in memory. Essentially, a data format has a data model and a representation scheme. Software interprets stored bits to create a memory representation of the modeled data ("read"), or interprets memory representations to generate stored bits ("write").

There are, of course, many possible data models, and many possible representations for a given data model. So, there many data formats have been created. Some have formal models with detailed specifications and universal standardization (e.g., DLL [34], netCDF [61, 62], JPEG [24], PNG [14]), others are informal, minimally specified, and used in a small niche. The latter includes output from programs, as well as forms and spreadsheets.

Relational databases support many data models, mapped to relational tables and operations. The SQL language enables a data model to be defined in machine-readable *schema*, which both documents the data model and enables software automation.

The XML language is designed to bring the advantages of a standard data model to a file based data format. XML defines a universal standard data format:

- A data model (the XML Infoset) [12]
- A mapping to storage (based on Unicode) [63]
- A schema language, XML Schema [67, 69]

This combination has made XML extremely successful means for data exchange. It is possible to transform XML to XML via XSLT [64]. The schemas fully define the data formats, and the XSLT stylesheet(s) define the transformation(s). Other standards define queries for XML ([4, 7]).

Because of its universal acceptance and technical advantages, XML is widely used for data exchange. Most relational data base systems provide import and export from and to XML (e.g., [42, 48]). Many software packages define mappings to and from XML (e.g., NetCDF Markup Language (NcML) [62], and HDF5 [28]). Tools such as Microsoft Excel™ also feature import/export to XML (e.g., [35, 36]).

The goal of the Data Format Description Language is to extend the advantages of using XML to data stored in non-XML formats. To achieve this, data is converted from its native format into an XML Information Set (i.e., a data model expressed in XML).[1] Once the data is represented in an XML Information Set, it can then be written to XML and/or transformed via any XML-enabled tool.

To achieve this, DFDL extends the XML Schema Definition Language [13, 51]. The logical data model is described in an XML schema document. Then the schema is augmented with DFDL annotations which describe the native representation of the data. For example, the XML schema might define an element called "weight", which is restricted to be an integer (the logical data). The DFDL annotations would define the location and layout of the integer in the native file format (the physical representation).

2.1. Procedural versus Declarative

One of the intellectual contributions of the DFDL is to define a purely descriptive method for translating data into XML (and back). The logic of translating to and from XML can be represented in code (procedural), a description, or a combination of code and description. Many computer problems and languages have a similar range of potential implementations. For example, one may solve a problem with a precompiled library (procedural) or an interpreted script (descriptive).

---

[1] In addition, DFLD enables a conforming XML document to be transformed into the original native format, which is termed "unparsing".

Trade offs between procedural and descriptive methods are well known from the design of programming languages and software. A procedural implementation can be compact and efficient, but may be opaque and fragile. A procedural implementation may be easy to implement, but difficult to port or maintain over time, because of its opacity and dependency on other software.

In contrast, a descriptive implementation is often less efficient in both storage and execution time than a procedure, but is more transparent and portable. A descriptive implementation may be more difficult to create, because the semantics must be explicitly written out. However, a descriptive implementation is easier to port and maintain precisely because the semantics are written down.

The DFDL is a purely descriptive approach: the specification and the annotated schema document are sufficient to implement the transformation to XML. In principle, DFDL can be interpreted by any software that conforms to the standard. Accessing the data does not depend on any specific software implementation, and it is possible to exchange data among heterogeneous systems without prearranged agreements. For any given case, one could write a program to transform the data to XML (a procedural implementation) or write a DFDL description. The program may be easier to write that the DFDL schema, and might well execute more efficiently than the DFDL parser. However, the DFDL schema will be easier to port and maintain over time, compared to a program.

2.1. Semantic preservation

If the physical data is preserved, and if the logical structure is preserved, and the data can therefore be accessed, is that sufficient? Unfortunately, the data may be useless if critical relationships and assumptions are lost. Problems may be as simple as dangling references (URLs or citations) or undocumented facts (such as units of measurement). These missing data can render data difficult or impossible to interpret.

For example, consider a data set with two fields, a date and a temperature. The DFDL description would map these fields to XML data structures (tags and values), and would give them standard data types (a date type and a number type). The semantic extensions enable recognition and extraction of relations such as:

- "*date* is the *date the reading was recorded*"
- "*temperature* is in units *degrees C*"
- "the reading was from a *sensor of type Z*"

Often, these relations may not be explicitly stated in the stored data, in which case extraction requires inference, and the metadata must maintain its relationship to the stored data.

These challenges to the interpretation of data stem from assumptions and relations that were implicitly recorded in software, or exist only in human knowledge. If the original software and people are no longer available, it may be impossible to use the data. Note that

this problem is relevant both to preservation for the future and for reuse of data outside its original context.

Essentially, DFDL is necessary but not sufficient to assure access to data. There is a need to capture and preserve a "semantic" description of the data. These relations and assumptions can come form the data itself (e.g., stored metadata), fro other data (e.g., provenance records), from repositories of expert knowledge (e.g., information about the implicit semantics of particular software), or from human knowledge. These semantics can be expressed in RDF.

3. Previous Work

The DFDL emerged from a history of many similar, but incompatible, unsystematic, and incomplete efforts. The DFDL specification builds on these experiences to create a single, widely-accepted standard.

Historically, there have been many data description languages, used for the specification of systems and for data transfer. The ISO ANS.1 standard is a complete data description language, capable of describing any data [22]. Data modeling standards such as SQL [44] have developed very complete languages for describing data. In the case of the SQL standard, there are standard conversions between SQL and XML [21], and implementations for most database systems (e.g., [42, 48]).

With the emergence of XML and the XML Schema language, XML has become the lingua franqua for exchanging and manipulating data. Because of its universal acceptance and technical advantages, XML is also widely used for data exchange.

Nevertheless, there remain good reasons to continue to use non-XML storage formats, including efficiency, security, and the requirements of specific types of data (e.g., complex linking patterns). Today almost every system needs the ability to import and export data formatted in XML. Widely used systems such as Database Management Systems and spreadsheets such as Microsoft Excel™ provide simple import/export mechanisms to read/write XML data as well as their native formats [35, 36, 42, 48, 49].

For systems that do not directly support XML, it has become common to create a logical mapping from a data format to a logically equivalent XML schema, and then implement tools to read the data and create XML, and to read XML and generate logically equivalent non-XML. Many data formats have developed mappings to and from XML, along with tools to implement these conversions. For example, netCDF [61] has defined an XML mapping [62], enabling translation to and from netCDF and XML.

It is conceptually possible to map almost any data structure to an equivalent XML structure, and to define an XML schema to define valid XML that can be translated to a given non-XML format. However, developing these mappings and related software often is a very labor-intensive process, and maintaining a plethora of readers, each useful for a small set of cases, is difficult and may ultimately be unsustainable.

It is clear that it would be beneficial to have a general-purpose system that takes a data description and generates software to generate (or ingest) XML from a given dataset. There have been many attempts to build such a capability, usually for a particular class of datasets. The Binary Format Description (BFD) language, was based on the Extensible Scientific Interchange Language (XSIL) [6], a language designed for processing scientific data, including multiple streams and arrays [41]. BFD employed XSLT to generate arbitrary XML data. BinX was a similar parser, which used description of the content, structure and physical layout (endian-ness, blocksize...) of binary files to generate XML [71, 72]. Many other examples have been created, including the OIL language [25], the Universal Parsing Agent [73], and commercial efforts.

These efforts demonstrated the concepts, and the implementations were successful within limited uses. From its inception, the DFDL working group sought to generalize and improve these efforts [13].

The DFDL project at NCSA was conducted as part of a larger project ("Innovative Systems and Software: Applications to NARA Research Problems"), and DFDL can be seen as a significant extension of other work on data format conversion. Format conversion is a knowledge-intensive process which requires several tasks, including:

- object characterization (i.e., type discovery)
- format specific encoding and decoding, and
- logical mapping of semantically equivalent data structures.

Bajcsy et al. have described a system that captures such knowledge for many "3D data formats", using a diverse collection of software integrated in a single service interface [3, 31]. The DFDL provides an alternative, explicit representation for the knowledge encapsulated in the components of this system.

The DFDL can implement many-to-many format translation via intermediate representations in XML. Using the semantic extensions to Defuddle, the transformations may also extract metadata into RDF, and thereby link to other relationships not explicitly represented in the stored data. These and other use cases are discussed in section 5.

The DFDL description can also be used for object characterization, e.g., MIME-type discovery. JHOVE [1], PRONUM [60] and similar systems provide a capability to discover the type of an object based on heuristic rules. This is often done via pattern matching or other simple rules which are encoded in executable code or services. We have shown that DFDL descriptions can be used to describe simple recognition rules (e.g., patterns to match). The descriptions are explicit and transparent, compared to code or scripts. In addition, if we use the semantic extensions, the content type is produced as RDF metadata, which itself is highly descriptive and portable.

4. The Open Grid Forum DFDL Working Group

Earlier efforts discussed above led to the establishing the Open Grid Forum working group to create the Data Format Description Language (DFDL). The goal of the activity is to create a complete standard to subsume the previous, non-standard, description languages and parsers. Unlike special purpose languages, DFDL is designed as an extension of XML Schema language [67, 69]. The DFDL Specification is now a Proposed Recommendation of the Open Grid Forum [13, 51].

Strict adherence to XML Schema is a critical design feature of the DFDL. The XML schema language is well designed and extremely general. Also, XML schema provides a universally accepted data model, and, because of its universality, there are many excellent tools which can easily use DFDL.

The DFDL effort received significant expertise and contributions from an international group of experts from IBM and other organizations, as well as NCSA. Several implementations informed the development of the standard, notably Virtual XML Garden [54] and Defuddle (discussed below). After five years, he working group released version 1 of the specification document, which has been designated an Open Grid Forum Standard [13, 51].

In addition to the standard document itself, tutorials, test suites and reference implementations are under development by the working group (for current activities, please see [13]).

4.1. The DFDL Standard Document

The DFDL standard document ([51]) is the primary reference for DFDL implementers. At more than 150 pages, the specification is quite complex. It defines the syntax for a set of annotations to an XML Schema document, along with the behavior of parsers that implement the standard. The document makes detailed reference to the XML Schema standard, and follows XSD terminology and syntax as much as possible.

Framing DFDL as an extension to XML schema required significant complexity, and constrained the semantics of the DFDL languages. A DFDL schema must follow the structure of an XML Schema document, which is always a *tree of nodes*. DFDL must define the semantics for all cases that may arise in XML schema, and may not use semantics that violate the structure of an XML schema document.

The specification calls attention to the following key limitations:

> "[…] DFDL does not intend to provide a mechanism to map data to arbitrary XML models. There are two specific limitations on the data models that DFDL can work to:
>
> 1. DFDL uses a subset of XML Schema, in particular, you cannot use XML attributes in the data model.

2. The order of the data in the data model must correspond to the order and structure of the data being described. " [51], p.10

The latter constraint is extremely important because it both simplifies the implementation of parsers, and limits the scope of a DFDL schema. Specifically, this means that a DFDL schema is intended to describe the layout of data, not to describe an abstract logical model into which the data is to be transformed. Fortunately, this limitation is at least partially mitigated by the fact that the resulting XML can be transformed by XSL or other tools. So, it is usually possible to create whatever XML is intended, though it may require two steps if the data must be rearranged.

In addition to defining the syntax of the DFDL languages, the specification defines the parser as "a recursive-descent parser [RDP] having guided, but potentially unbounded look ahead that is used to resolve points of *uncertainty*." [51], p. 55 (emphasis in the original). This logical parser is defined in detail, along with the syntax of the language and rules for resolving uncertainty.

In addition to the specification, the working group has developed a tutorial for the language and is developing an initial test suite [13]. To date, no reference implementation is available. In later sections we discuss our experience implementing the standard, and sketch the basic design.

5. Why: DFDL Use Cases

The DFDL is not a simple solution to a single problem, it is a general, foundational technology that has many potential uses. This section describes some of the key use cases, as well as some cases where DFDL should *not* be used, and what DFDL does not.

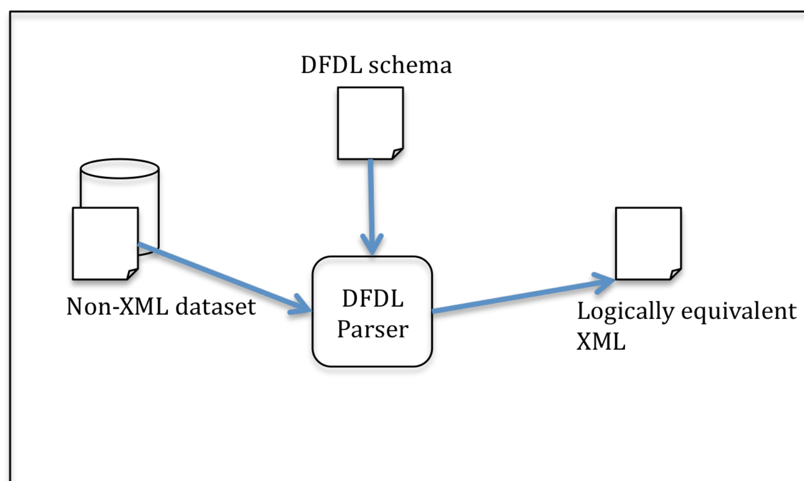

Figure 1. General use of DFDL

5.1. "Get the data into XML"

A core use for DFDL is to "get into XML", i.e., to enable the plethora of XML-enabled tools and systems to access and manipulate data from non-XML sources. As already suggested, the XML representation of a dataset is amenable to manipulation through standard tools, as well as import into many programs, and output in many formats.

Figure 1 illustrates the general use of DFDL, to create logically equivalent XML. This section sketches some of the use cases for this capability

5.1.1. Interoperability and portability

The declarative style of DFDL is a great advantage for interoperability in a decentralized system. In order to share data stored in a native format, it is necessary for the receiver to read the data. In a decentralized system, such as the Web or a Grid, there is no guarantee that the receiver has the requisite software and knowledge. One approach is for the receiver to obtain and install a reader from a library of modules, or use an online service that implements this step (e.g., [33, 70]). This approach requires creating and maintaining the library of reader software, adding new readers for every new data format in the system.

In this case, DFDL has the advantage that the description itself is portable and platform independent. To read the data, it is not necessary to preserve and port the reader software, only to interpret the DFDL description with a conformant parser. DFDL can be used to generate data readers as needed, rather than requiring the reader software to be located and ported. This is especially valuable for cases where there is no standard reader available, such as data written from simulations or specific instruments.

The DFDL description itself is an XML schema, and therefore potentially accessible via a URL. Instead of managing the distribution of software, one can point to a (non XML) dataset and provide the URL of its DFDL description. This combination is sufficient to access the data. Figure 2 sketches one such scenario.

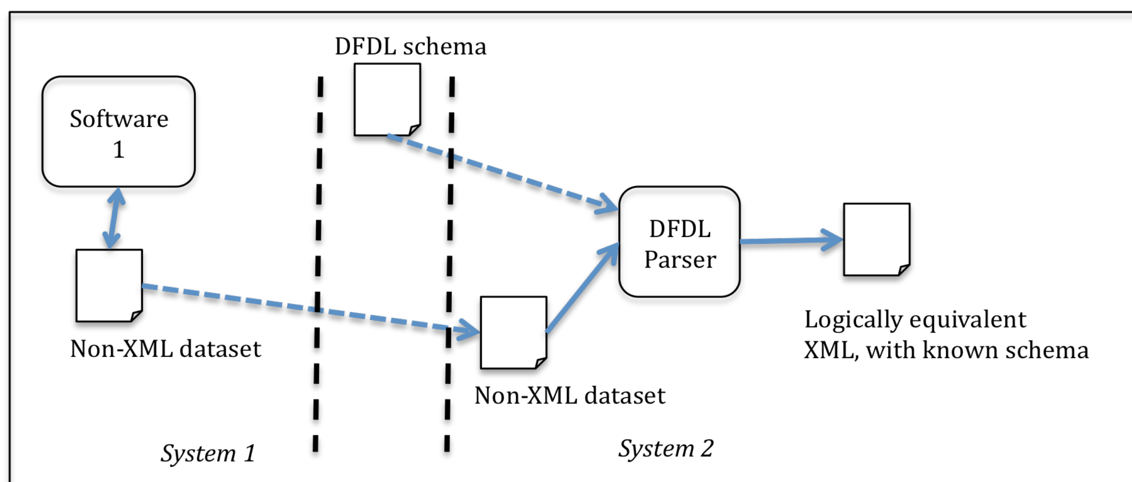

Figure 2. Interoperability in a Distributed System

Once the data is extracted into XML, it is available to *any* XML capable tool. A user can access the data without any knowledge of the native format at all. (Obviously, the receiver needs to understand the *meaning* of the data, as discussed in later sections.) In addition, this reduces the difficulty integrating the reader into existing systems or infrastructure, assuming that the system already handles XML.

5.1.2. Translation: XML as Lingua Franca

Interoperation of data often requires transforming data from one format to another. While it may be possible to find a chain of conversions between two formats (e.g., [33]), given the infinite variety of possible data formats, it will never be possible to create conversion software for all combinations. DFDL partly ameliorates this problem by enabling transformations via intermediate XML formats.

Defining a DFDL specification for the data will enable the data to be read into a known XML format, from which it can be transformed to any other supported format. DFDL provides the key bridge from any data format and the interoperable space of XML data. Transforming data from one format (A) to another (B) is possible when they are logically compatible. Transforming A to B can, in principle, be accomplished by transforming format A to XML, transforming the XML to different XML if necessary, and then transforming the XML to format B. Figure 3 sketches this scenario.

In this process, it is still necessary to convert the XML representation of format A to the XML representation of format B. In general, this mapping is considerably easier between two XML schemas than between two native formats.

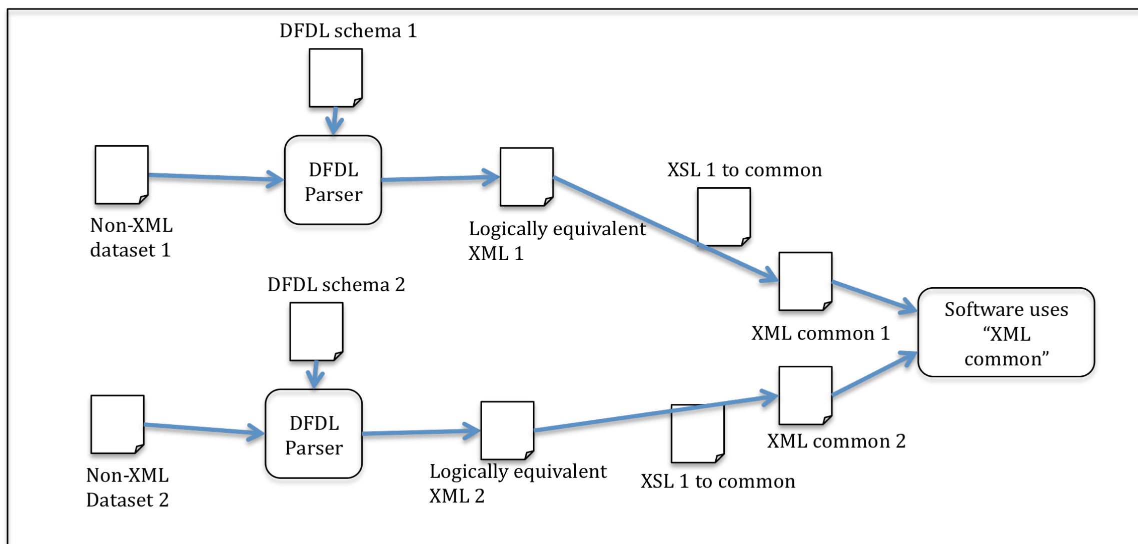

Figure 3. DFDL enabled translation to a XML format.

5.1.3. Extraction and Combination of Data

One of the important features of XML is the ability to address data items within a single XML information set, e.g., with XPATH and XQuery [4, 7]. DFDL makes it possible to extend this facility to non-XML data. The data would be accessed in two steps, applying the DFDL description to generate XML, and then applying the XPath to identify the data item. This could be done on-demand, i.e., the data would be stored in native format, and the relevant transformations triggered only when the data item is needed.

The DFDL enables interesting possibilities for manipulating datasets. One interesting possibility is the construction of "synthetic datasets", which combine data from multiple sources, possibly in multiple native formats. Essentially, the synthetic dataset would be constructed by selected data from one or more XML files, to create the 'synthetic' file. DFDL descriptions would be used to extract the data from several native datasets.

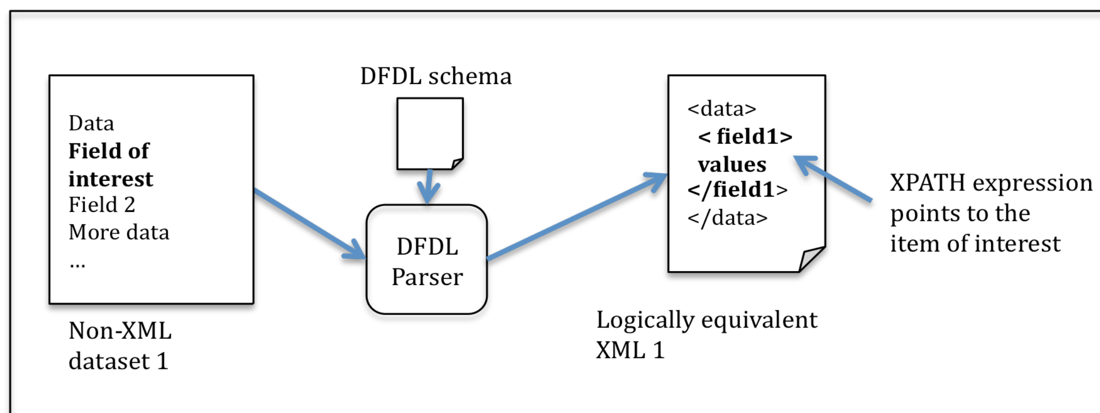

Figure 4 Extraction of metadata

5.2. Data Preservation

Archiving and preservation of digital records faces difficult challenges in handling data formats that are not well-established standards. While the bits of the data might be preserved, if, at a future time, the software that created the data is no longer viable, it may be impossible to access the data. Even if the data is still readable, semantics may still be lost.

DFDL can play an important role in the long-term preservation of access to data. Just as the DFDL description enables interoperation across systems, it also enables interoperation across time.

In recent years, large-scale environments for digital preservation have emerged. In the US, The National Archives Center for Advanced Systems and Technologies (NCAST) Transcontinental Persistent Archives Prototype (TPAP) testbed has demonstrated a complete end-to-end infrastructure [37, 43]. In Europe, the SHAMAN project has developed an environment for digital preservation, including access to data [70]. Both these projects build on iRODS data grid technology. The Networking and Information Technology

Research and Development (NITRD) sponsored a large scale demonstration which connect TPAP, SHAMAN, and other sites [38].

As part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, a suite of tools an services have been developed that address aspects of these problems ([3, 32]). Designed as RESTful services, these tools are available over the Web, and can be "mashed up" into many different systems [20].

These environments implement powerful processing capabilities, including rule-base automated workflows [39, 70]. This infrastructure enables access to data by recognizing specific digital objects (e.g., files), and applying appropriate software to access the data, possibly by translating to a form that can be accessed. These workflows provide a persistent knowledge of how to access data.

Of course, sustaining this approach requires maintaining the software required for access, along with any environment and dependencies needed. These approaches can be seen as primarily "procedural"—to access object "A", the system must invoke "ReaderForA". The preservation environment must keep all necessary modules viable for as long as the data is held.

The SHAMAN project addresses software preservation by writing the modules in Java which executes on the Java Virtual Machine. The hypothesis is that the Java Virtual Machine can be maintained in the future [11]. This is reasonable, since Java software will be viable as long as a conformant VM is available, which is much simpler than maintaining operating systems and system libraries.

DFDL is a promising technology for this use in these systems. To access data using a DFDL description requires only a conformant DFDL implementation. The DFDL schema both documents the logical structure of the data in a standard form (XML Schema plus DFDL), it is sufficient to enable creation of access software. The DFDL software can transform the data into XML with a known structure, from which it can be transformed to whatever form is needed.

The formal specification, and the reliance on the XML standard, should make it possible to independently create a parser. That is, it is not necessary to preserve today's parser implementation and its context, only to preserve the data and the DFDL description, plus the relevant standard specifications. This model should significantly reduce the amount of machine and operating system dependent software that must be maintained to preserve access to file content.

Since DFDL schemas are XML schemas, they can be addressed by URIs. This means that archive systems can maintain collections of authoritative DFDL descriptions, which can be accessed via the network.

5.3. When Not to Use DFDL: Anti-Use Cases

DFDL has an important role in accessing non-XML data, but it is not a universal tool for any data. First, *any data that already has a schema does not need DFDL*. Clearly, DFDL does not apply to XML data, XML Schema and other tools already provide the capabilities. In addition, *DFDL is not needed for relational databases*, because these already have a schema. Usually, relational database management systems also have tools to generate or ingest XML when needed.

When considering the application of DFDL, it is important to realize that DFDL describes *data*, not *algorithms*. There are many important storage schemes that require algorithms to interpret the data. Foremost among these are compression schemes, which apply an algorithm to the stored data to recover the stored value. Other examples include hash tables and B trees, and various object storage formats (e.g., HDF5 [59] and STEP [56]) which also employ algorithms to interpret the stored data.

In these cases, the dataset has multiple layers of semantics, the layout of the stored bits and one of more overlaid interpretations. For example, a "gzip" compressed file is stored in a stream of bits and bytes. These bits are interpreted as input to the compression algorithm which creates another set of bits which is the intended data values. DFDL can describe the storage layout, and transform the data into logically equivalent XML. But, DFDL cannot specify how that XML is interpreted (e.g., as input to a particular decompression algorithm), nor can it describe algorithms.

For example, imagine a block of compressed data. The DFDL description will define the block of data as a block of numbers, which a DFDL parser can extract into XML. However, these numbers are meaningless without the application of the relevant decompression algorithm.

Finally, some data formats define *linking and pointer structures* that enable references within a single file. XML itself is structured as a tree which is linearized in its text representation. As a consequence, it may be difficult or impossible to model certain kinds of linking with XML, notably general graphs or meshes. For example, a file that contains a B-Tree is difficult to represent in XML, because the storage has multiple references forward and backwards, to locations in the file. HDF5 ([59]) and PNG ([14]) are characterized by such structures, and therefore are poor candidates for a DFDL description.

5.4. DFDL is *Not* Artificial Intelligence

DFDL is sometimes imagined to be a general-purpose data processing system, capable of interpreting arbitrary data. This is not really true, though DFDL is a valuable piece of such a system.

The DFDL is a language for representing knowledge about the logical structure of data and, in conjunction with XSLT and other tools, can represent some data transformations and translations. The semantic extensions provide tools for representing knowledge about the

semantics of data. Furthermore, the DFDL standard enables software to be automatically constructed to parse and unparse data.

A DFDL parser cannot, however, *discover* the logical structure of data or its semantics. This information is encoded in the annotated DFL schema document and XSLT. Creating these schema requires human understanding, and can be quite difficult if the data is complex. Creating the DFDL schema is labor intensive and requires explicit declaration of all the details and contingencies in the logical structure, which can be complex.

The DFDL does not eliminate or replace this human effort, but it does help the data modeler in at least two ways. First, it provides an open standard for representing the results of this effort which is machine readable and portable. This enables the maximum return for the invested effort. Second, DFDL support can be added to data modeling tools (e.g., the Eclipse XML schema editor [58]) to extend their value and ease the difficulty of creating DFLD schemas.

II. Practice: Implementing DFDL Parsers

6. DFDL Parser Design and Prototype Implementations

The DFDL language is expressed in a set of annotations on an XML Schema, which follow the XML Schema specification. This section discusses the general design of a parser for the DFDL language.

Over that past several years PNNL and NCSA have implemented two prototype parsers. Section 6.2 describes the current prototype is Daffodil [52]. Section 6.3 describes an older, unsuccessful parser, Defuddle [30, 57].

6.1. Data Format Description Language Parser Architecture

The Data Format Description Language (DFDL) Specification defines a language is expressed in a set of annotations on an XML Schema, which follow the XML Schema specification [30, 57]. Note that a valid DFDL Schema must be a valid XML Schema, although the DFDL annotations are only meaningful to a DFDL-aware Schema Parser.

DFDL parsers accept a DFDL schema and input data, and produce XML output data.[2] DFDL is a subset of XML Schema that includes annotations that describe how target XML elements correspond to structures in the input data format. The annotations are used to control a DFDL processor, which consumes the input data and produces XML.

The DFDL parsing process has three phases, Schema parsing, Data processing, and output (Figure 5).

[2] DFDL also defines "unparsing", i.e., reading an XML file and generating the corresponding data file. "Unparsing" support is an optional feature, and will not be discussed in this document.

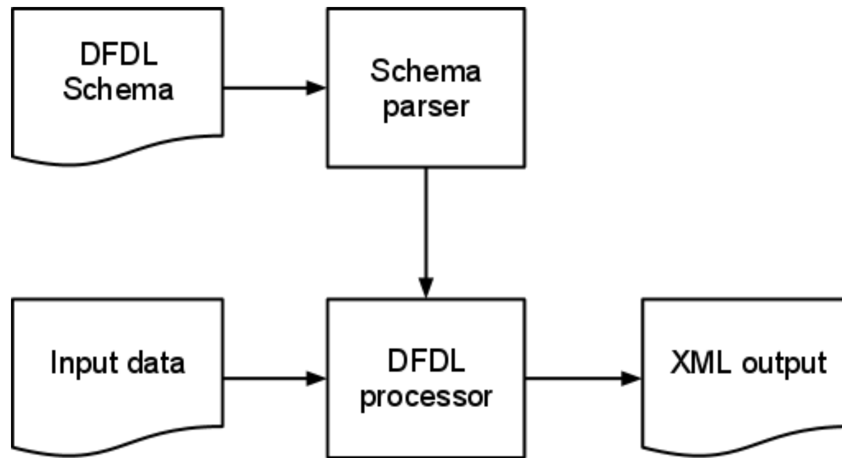

Figure 5. DFDL dataflow.

The Schema Parsing phase must analyze the DFDL Schema as per the rules of XML Schema (which define the XML Info set) and also identify any DFDL annotations, which define the behavior of the parser. The DFDL specification defines the semantics of the annotations as well as scoping rules which define the context within which the directives are to be applied. As noted earlier, by definition, a valid DFDL schema must be a valid XML schema.

The Data processing phase reads data from the input file, interpreting the text or binary data according to the DFDL annotations, and creating the XML Information set, populated with values read from the input file.

Finally, the XML Information Set is output, e.g., as an XML file. Note that the resulting XML may be processed and transformed through any standard XML tools. One example of such processing is the application of GRDLL, which extracts metadata and generates RDF (discussed in later sections).

These abstract phases can be implemented with different technologies and approaches. The Schema Parser may generate code for a data parser, or may guide the execution of a general data processor, or some combination.

The DFDL specification defines required features, and the constraints of the specification impose requirements on the implementation, particularly the need to maintain the global and local context in order to correctly handle default values. In addition, the DFDL specification limits the Schema such that, "[t]he order of the data in the data model must correspond to the order and structure of the data being described" ([51]. , p.10). The latter constraint ensures that the parser does not need to support arbitrarily complex rearrangements of data. With this constraint, the input data usually can be treated as a stream.

The DFDL specification also defines the "unparsing" behavior—given a DFDL schema and conforming XML, create the logically equivalent non-XML dataset. For many simple cases,

this is straightforward, but there are subtle issues which make a complete implementation challenging. Essentially, the “logically equivalent” XML does not necessarily contain all the information necessary to reconstruct the original input data.

For example, if a text file has lines that may be terminated by either ‘\n’ or ‘\r\n’, the XML representation of the (logical) data would not record which terminator was used. The DFDL unparser would know it was one of two, but not which one. It would be possible to create a “logically correct” dataset, but not to reproduce the original dataset, if that were needed.

This and similar cases will need to be worked out for any complete DFDL unparser. For example, there could be a “hidden” record of the original layout of the data, either in an auxiliary file or in annotations in the XML.

Note that it may or may no be necessary to reconstruct the precise original file, bit-for-bit. Many uses may be satisfied by a logically equivalent dataset. However, for long term preservation of data, it is usually required that the system be a loss-less as possible. Also, some processing scenarios require the ability to reconstruct the original data, e.g., for conformance to other software requirements.

6.2. The Daffodil Parser

The Daffodil parser ([46]) is a partial implementation of concepts from the DFDL specification, written in the Scala programming language [15, 47]. Daffodil is a fresh implementation, it does not build on any previous code.

Daffodil is an experimental codebase that provides implementations of some of the concepts and ideas from the Data Format Description Language specification. *It is not a correct implementation of the DFDL specification,* although its behavior and overall architecture resembles what such an implementation would be.

The Daffodil codebase could be used to develop an implementation of DFDL, although there are major architectural components that would need to be extensively rewritten with close attention to compliance with the DFDL specification. It is possible that some of the text and binary processing code could be reused, but that will need to be examined further before a determination could be made.

Daffodil's entry point accepts configuration parameters, locates the input schema and data, parses the schema to construct an internal representation of the XML Schema components and their DFDL format annotations (in a standard Document Object Model, DOM), and then invokes the DFDL processor on the input stream. After producing XML, it optionally applies a GRDDL transformation to produce RDF output (for our earlier work using of GRDDL, see [30]).

Figure 6 shows the main classes in Daffodil.

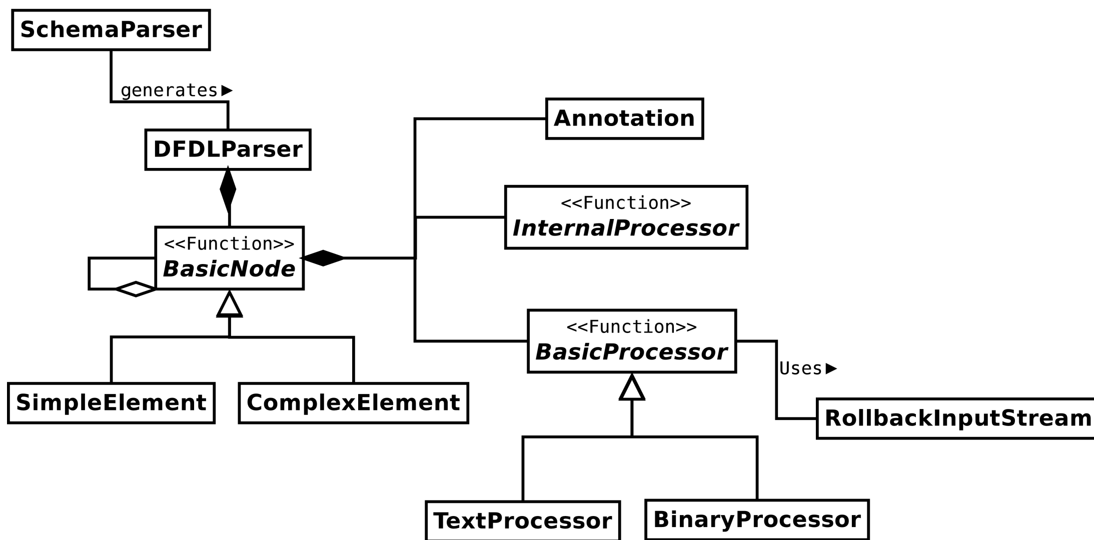

Figure 6. Daffodil architecture. (Note that this is slightly out of date: DFDLParser is now called AnnotationParser.)

6.2.1. Architecture components

Schema parsing is accomplished via the SchemaParser and AnnotationParser classes. SchemaParser descends the DOM of the DFDL schema, invoking AnnotationParser on annotated schema components to produce Annotation objects. Each Annotation object is then used, along with contextual information, to instantiate sub-parsers called BasicNodes. Each BasicNode is associated with the corresponding DFDL schema component.

The BasicNodes are, essentially, the implementation of the input processing phase.

6.2.2. Input Processing

Input data processing is divided among BasicNodes. BasicNodes are functions that are applied to input data. At the entry point (i.e., the beginning of the parse), the BasicNode corresponding to the DFDL schema's root type is applied to the input data.

When a BasicNode is applied to input data, it generates one or more DOM Elements, instantiates and invokes a Processor to produce output data, and inserts the output data into the DOM Elements generated. As part of this process it may invoke child BasicNodes, which causes the elements they generate to be added as children to the DOM Element associated with the enclosing BasicNode. The result at the top level is a single DOM Element, representing the entire input data. The top level DOM element is serialized into an XML file to produce the result.

A variety of Processor classes are provided that interpret input data based on DFDL format annotations. Other kinds of DFDL annotations are handled within BasicNodes, in order to support DFDL features such as hidden elements, assertions, and some forms of reference.

Since each BasicNode is self contained and generic, the process phase is cleanly separated from the schema parsing, and should work the same for all input data. It is the responsibility of the schema parsing phase to instantiate the correct set of BasicNodes, with correct parameters and context. In turn, it is the responsibility of the BasicNode to produce a correct DOM node (or nodes) when invoked, for any schema.

In response to lessons learned from the Defuddle parser ([29]., and below), Daffodil implements basic memory management for the DOM tree. The goal is to enable portions of the tree to be "paged out" of memory while other parts of the tree are parsed. This capability enables Daffodil to handle larger files without exhausting memory.

6.2.3. Implementation status and issues

The Daffodil parser implements something very much like DFDL, but it does not comply with the DFDL 1.0 specification. There are issues with its schema parsing, and its code-level documentation, that make it difficult to assess what needs to be done to achieve compliance. The following outlines the issues and a plan for addressing them (see also, [17]).

The Daffodil source itself contains test cases, termed "internal" tests. In addition, the OGF DFDL working group is developing a test suite designed to verify that output is correct given specific schemas and input documents constructed to exercise various features of DFDL itself. In early 2011, this suite was under development, and we used an early release of the tests. Daffodil does not pass any of these tests, due to issues discussed below.

There are several areas in which the internal test schemas, and the Daffodil implementation, differs from the standard:

1. Daffodil appears to implement property scoping incorrectly
2. Widely-used, required features, such as default formats, are not supported
3. DFDL representation property names, and controlled vocabularies for their values, differ between the DFDL specification and Daffodil. (I.e., items and values have incorrect names).

The test suite from the OGF DFDL working group consists of schemas that require these features. Because of the issues listed above, Daffodil fails all the OGF tests. It is impossible to know what other features may fail until these issues are resolved.

In summary, the test suite and inspection of the internal tests and code show that Daffodil does not comply with the DFDL specification in many ways. While mis-named values can be addressed easily, the issues with scoping and defaults are significant.

It is difficult to tell if Daffodil's text and binary processing facilities comply with DFDL, because they must be configured with schemas, and problems with the schema parser make that difficult.

Daffodil has very little code-level documentation. Scala's dynamic language features, and the paucity of robust development tools for Scala makes it difficult to perform static analysis to determine how the code is organized and functions. Automated documentation via the scaladoc tool helps, but provides only the skeleton of documentation, since scaladoc comments are missing from most of the code.

Furthermore, although Daffodil implements a specification, it does not identify which version of the specification it implements, nor is its code associated via code-level documentation with the relevant sections of the spec (for example, referencing the definition of a schema annotation from a comment in the code that parses it). Both of these problems can be addressed now that version 1.0 of the specification has been finalized.

7. The Defuddle Parser: An Unsuccessful Implementation

The Defuddle parser was created at PNNL in 2005, based on a snap shot of the DFDL standard at the time [57]. It is important to note that, at the time Defuddle was created, the DFDL standard was nowhere near completion, and many aspects of the DFDL were changing drastically from month to month.

Defuddle was intended to demonstrate the concepts of DFDL, and to try out features that might be included in the specification. It was hoped that Defuddle would ultimately converge to the standard as it was finalized, but that was not the primary goal of the initial implementation.

The implementation strategy was to build on the open source Apache project JAXB, which became JAXME [2]. JAXB/JAXME is an XML parser generator, implemented in Java. Given an XML schema, JAXME generates Java classes to parse and validate documents that conform to that schema. The output of the parser is a DOM tree (in Java data structures), which is output as XML.

Defuddle was built by modifying the JAXB source to recognize DFDL-like annotations, and generate a parser which reads the input file (non-XML) and generates a DOM tree which is usually written out in an XML document. Since there was no final DFDL specification at the time, Defuddle implemented a set of annotations that were faithful to the concept of DFDL, but ultimately had little correspondence to the final specification.

The Defuddle parser was taken up at NCSA in 2006, where it was extended with GRDDL. ([30] and discussed below). We also evaluated the performance of the parser, as well as fixing bugs and porting from JAXB to JAXME.

7.1. Performance Investigations

We conducted several performance tests for the Defuddle implementation [29]. While Defuddle did not implement the final DFDL specification, the overall performance results

might be relevant to any similar parser, and were used to guide the design of the Daffodil parser discussed earlier.

The performance studies determined that the critical performance factor was the use of memory during the parsing phase. Given that the parser is automatically generated code run in a Java Virtual Machine, these issues are highly dependent on the specific implementation of the JVM. It should be noted that this issue is not a matter of CPU speed or disk IO, and is not addressed by parallelization.

One of the key issues is that the parser is constructing an XML Information Set, which is conceptually a tree of elements and attributes. The JAXME generated parser represents the tree in Java DOM, instantiating Java objects for all the elements of the tree, and holding the tree in memory as it is constructed.

The size of the tree in memory is a critical issue. Since each node in the logical data structure is a Java object, the representation in memory is substantially larger than the raw input data that is represented. For small files, this does not matter, but larger files soon fill memory with the partially build tree, forcing the Java VM to garbage collect, and the operating system to page. In fact, even moderate size files exhausted the memory available to a 32-bit Java VM (~1GB of heap space). When the memory is exhausted, the parser will fail at that point.

The size of the tree depends heavily on the design of the schema and on the input file. A schema with deeply nested elements will have many more nodes than a "flat" data model, and a larger input file will generate more objects in memory.

It should be noted that this issue is produced by the behavior of the standard Java implement of the DOM tree. The Daffodil implemented a more scalable "paging" scheme, to allow the DOM tree to be larger than the available memory.

7.2. Critical Flaws in Defuddle

When the DFDL specification entered final review, we considered the question of what would be required to make the Defuddle parser conform to the Version 1 specification [53].

Building on the JAXME code base had obvious merits. It gave us a working parser to start from, with the infrastructure to dynamically generate Java code for the parser. However, these advantages came at a steep price.

First, the source code is complex (> 100,000 lines), poorly documented, and an open source community has not emerged. Maintenance and extensions of Defuddle required heroic efforts to understand and reverse engineer the often buggy and incomplete code. Even the smallest changes were nearly impractical.

Second, the JAXME parser is designed to parse XML Schema, and to generate parsers for XML documents. The specific design of the parser algorithms in the schema parser and generated parser code is suitable for the XML languages.

Examining the JAXME parser, it is clear that there is no way to implement many of the key concepts of DFDL in the existing code. For one example, in JAXME the data structure which represents an XML Annotation lacks a pointer to its enclosing element. This makes it impossible to implement relations and constraints on DFDL Annotations which refer to the context of the Annotation.

Overall, the JAXME parser simply is not designed to implement the scoping rules required by DFDL. So, even if the practical issues could be overcome, extending JAXME is simply not a viable strategy to implement DFDL.

7.3. Conclusions About Defuddle

The result of our analysis was to abandon the Defuddle parser, and implement the DFDL specification in a new parser, Daffodil [53]. A new implementation offers the opportunity to correctly implement the DFDL specification, and to address the performance and scaling problems.

III. Extensions

8. Semantic Extensions to DFDL

Beyond maintaining the accessibility of the raw bits of the digital data, preservation requires maintaining an ability to interpret the data as meaningful structures, i.e., within a larger "semantic" context. For example, it does little good to know that a field is a integer with value 107, if it is not known if this is to be interpreted as "degrees C", "kilometers above sea level", or some other unit.

One of the primary motivations for the Data Format Description Language (DFDL) is to preserve access to the meaningful logical structures of data across systems or time. An XML schema explicitly describes the 'logical model' of the data, and the DFDL annotated schema defines the logical structure for data that may otherwise be undocumented or implicitly expressed, e.g., in the code of a "reader". While the XML Schema language is well suited for describing the layout of data (the "syntax"), interoperability and robust archiving require "semantic" mark up as well.

The XML Schema language is designed to describe the structure of data, but it cannot express all structures and relationships. Fundamentally, XML is a hierarchical data model, while logical models may have more general relationships. Further, XML describes a single "document", with identifiers that are local to the current document. This means that XML cannot interpret, for example, an association of logical model elements with external resources (e.g., to express annotations or links).

As part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, we explored extensions to the DFDL to address these issues. Specifically, the goal was to create a language and parser analogous to DFDL that describes the structure and relations in the data in standard Semantic Web languages (the Resource Description Framework (RDF) [5, 19, 26, 66] and the Web Ontology Language (OWL) [68]). In addition to providing a richer framework for defining logical models and supporting global identifiers, an RDF/OWL analog of DFDL would be more amenable to logical inference and the use of rules to automate further enrichment of logical models and their associations with other resources.

RDF is the best choice for this language because it is the universally accepted standard for semantic description and linking. Also, as in the case of XML, there is a plethora of tools and technologies that exploit RDF. Thirdly, the formal semantics of RDF assure that the output of this phase can be linked with, combined with, and operated on along with RDF from other sources.

It is natural to think about a language directly analogous to DFDL, one that maps data to RDF assertions (i.e., relations). Such a language is difficult to design, for many reasons. IN general, this would require a general model of data and relations, along with a schema to define mappings.

Our approach was to create a two-step process. The standard DFDL provides the necessary first step to generate XML, and a second phase to extract semantic descriptions from the XML (see [30]). This builds on the DFDL, which serves as the model for data. The second phase is only required to map data items defined in an XML schema document to RDF triplets.

This two-phase approach can be used with any DFDL parser, indeed, we implemented the feature in both Defuddle and Daffodil. In our experiments, the second phase was built using Gleaning Resource Descriptions from Dialects of Languages (GRDDL) [10] as a standard mechanism for declaring these transformations.

In summary, our semantically extended DFDL focuses on a standard XML schema, decorated by two types of annotations:

1. DFDL (with extensions) to describe the original data and create logically equivalent XML
2. GRDDL plus XSLT describe structures and relations in the XML data and create RDF to describe the relationships

This architecture is described in more detail below and in [30].

The RDF output from the semantically enhanced DFDL parser can be considered both as a file and as a collection of triples. GRDDL engines have different options for how the RDF is produced, i.e., how the triples are serialized.[3] However it is serialized, the RDF triples can

[3] The elements of RDF are an abstract collection of triples, of the form subject, predicate, object. The formal semantics of these triplets are defined by the RDF standard (e.g., [19]). The abstract triples can be "serialized" in

be used as part of a larger graph of information including, for instance, provenance, domain ontologies, and social network information. To support this, the RDF triples would be ingested into some form of triple store where they can be queried and used in subsequent logical inference, rule-based, and other processes.

8.1. An Example Implementation

As part of our investigations, we implemented a prototype of the two-step approach. The first phase was a DFDL parser (initially Defuddle, later Daffodil). The second phase used Gleaning Resource Descriptions from Dialects of Languages (GRDDL) [10], to apply one or more XSLT stylesheet to extract RDF from the XML. Figure 7 sketches the basic design.

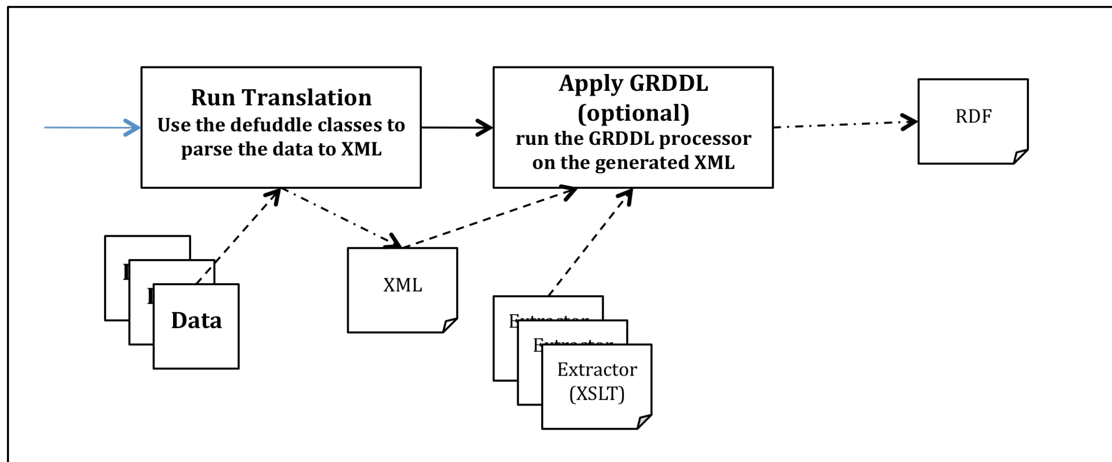

Figure 7. Basic operation of the GRDDL extension to Defuddle.

8.1.1. The GRDDL standard

There are several implementations of the GRDDL standard available [16]. The two most complete are GRDDL.py [18] and the Jena Plugin [23]. The former requires Python but is self-contained. The latter is part of Jena and thus is linked directly from Java.

Our initial implementation used GRDDL.py because it is more convenient than the Jena plug-in. Future implementations could make the choice of GRDDL engine a configuration option. In any case, the input files, DFDL and GRDDL markup, and GRDDL transforms are the same no matter which engine is used. The engines differ only in how they are invoked, how the output is provided, and potentially on robustness and performance.

GRDDL transforms are XSLT stylesheets that read XML and generates RDF. GRDDL does not specify the content of transforms except that they generate RDF. Note that any number of transforms may be applied to a given XML document, thus making it possible to consider each logical type/substructure in a model independently and to reuse XSLT files generated in the broad community. For example, transforms are available which process Friend-of-a-

---

several formats suitable for storage and transmission by computers (e.g., RDF-XML [26] or N3 [5]). No matter what format is used, the triplets have the same logical meaning.

friend (FOAF) records [8] which can be used to extract relations from any XML that contains FOAF records. Such a transform could be used in combination with those that extract bibliographic information from Dublin Core, transforms to scientific unit ontologies, and so on.

8.1.2. Integration with DFDL

The GRDDL standard defines assertions to declare the GRDDL name space and then list one or more "transformations" to be applied [10, 65]. The transformations are URIs that point to a description of a transformation. For our purposes, the transformations were XSLT [64] stylesheets that process the XML output of DFDL parser and generate RDF triples.

For example, in an XML file with the top level element "workflow", to apply a GRDDL transformation called "vistrails2rdf.xsl" (which we use to convert provenance created by Utah's Vistrails visualization software to an RDF binding of the Open Provenance Model [40]), the XML would be annotated in the following manner:

```
<workflow id="120" name="part1"
xmlns:grddl="http://www.w3.org/2003/g/data-view#"
grddl:transformation="http://vesta.ncsa.uiuc.edu/GRDDL/xsl/vistrails2rd
f.xsl">
```

In our initial approach, the DFDL parser inserted the required text in the XML it generates. This approach is simple and is known to work with current GRDDL engines.

In order to extract novel structures and relations of interest, it is necessary to create one or more XSLT transforms. Once transforms are created, they can be published at well-known URLs and used by any GRDDL engine, and hence may be used as community resources independent of DFDL. One of the central purposes of the GRDDL standard is to provide a declarative mechanism to document which transforms can be used for a given XML schema or instance and create a framework for community sharing.

8.2. Implications: Uses of the Extracted RDF

The semantic extensions to DFDL produce RDF which may have many uses. This section sketches some use cases.

8.2.1. Metadata Extraction

The semantic extensions can be used to create RDF to express metadata from the file. While metadata can be expressed in text, XML, or relational tables, only RDF directly and unambiguously expresses relations.

For example, the Dublin Core metadata standard defines common relationships such as "creator" and "date". This information may be present in a data file in some for. If so, then it can be extracted by:

1. Converting the data to XML via DFDL

2. Extracting the metadata relations from the XML with an XSLT transform to create RDF

This first step creates XML with the metadata in some of the XML elements, which may be specific to the particular dataset. The second step finds the relevant elements and generates RDF to express the relationships in standard form. These steps are implemented in a single call to the DFDL parser. Figure 8 illustrates this process.

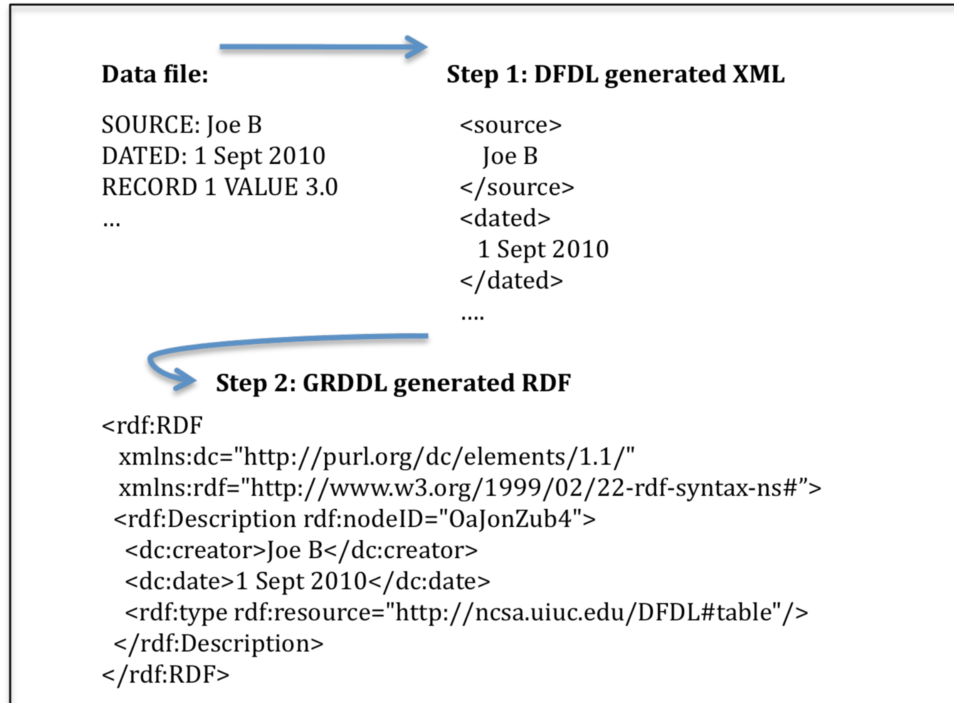

Figure 8. Two step process for metadata extraction.

8.2.2 Domain-specific Semantics

The XML data model used by DFDL describes generic "computer science" data types and structures—numbers, strings, nested structures, etc. Of course, real data usually describes domain specific structures, i.e., data organized according to the concepts of the particular knowledge domain of interest. For example, a three-dimensional model of an underground water system would likely be represented by a collection of variables and arrays, which are meaningful only when their relationships are interpreted correctly.

Consider a hypothetical dataset containing measurements of soil characteristics at various depths across a geographical area. This dataset might be represented at a three dimensional array of numbers. To correctly interpret these numbers, it is necessary to understand the units of measurement ("pH") and the relationships between the elements of the array and the measured world, e.g., what location and depth are recorded by each position in the array. The data set may contain additional data to record these facts, e.g., additional arrays that contain latitude, longitude, and depth for each data point, along with annotations to document the data.

To understand the "semantics" of this dataset, it is necessary to understand the relationship between these data items, as well as the relationship to theory and the real world. These semantics are often embodied in software code and human understanding (expert knowledge), but may not appear explicitly in the stored data.

Our work is based on the intuition that, for any given case, it is possible create an "ontology" of the operations for structures and relations. Such an ontology would build on standard concepts of data structures and relations, to implement descriptions of domain specific structures and relations. For example, a hydrology dataset could be described in terms of domain-specific objects that are composed of a specific collection and arrangement of generic arrays, strings, tuples, and so on.

RDF can explicitly represent these relationships. Thus, the extended DFDL enables a more complete description of the dataset.

8.2.3 Linked Data

More generally, semantically extended DFDL enables generic datasets to become part of the wider mesh of linked data.

It would be natural to implement a mechanism to invoke DFDL and GRDDL as part of a file "upload" operation, and to store the output as metadata or new content linked to the original data via provenance information. The result would be data in standard formats (RDF and XML) with additional references to standard schemas and ontologies. The RDF and XML can be further processed to transform the data structures and relations into whatever abstract forms are needed. In particular, the data can be transformed into data structures needed for analysis and visualization. The dataset can be linked to other data, and be linked to by other data via RDF and XML path expressions. The RDF can connects these metadata to ontologies describing the concepts for one or more domains those topics. Note that this adds relevant semantics to the logical descriptions that are not directly retrieved from the original file instance.

The linked data view enabled by DFDL augmented with RDF provides general mechanisms for processing data. RDF can store representations of operations in portable and machine-readable forms, and RDF would be amenable to automated reasoning (e.g., to guess appropriate default actions).

8.3. Limitations

Our two-phase approach is practical and can meet many needs, but has limits. Like any linked-data system, it is challenging to manage identifiers required by the RDF. The extracted RDF may refer to other items in the dataset, to the dataset itself, or, indeed, to any URI. Not only are identifiers difficult to create, the semantic extraction phase must have a mechanism for generating identifiers for all the objects in the data, and for any links outside the dataset.

To give a simple example of this problem, consider the simple case of creating Dublin Core metadata in RDF. The RDF will have an assertion relating the value of a data item as "dc:creator" of the dataset (the object of the RDF triple). What URI should represent the dataset? Is it the original dataset, or the XML generated by DFDL? And how does the metadata extractor determine the URI? This problem occurs throughout the process, since

the RDF may need to state relations between items and any URI, not limited to the single dataset.

These issues can be mitigated to some extent by using temporary identifiers (chosen to be unique within the context), and post-processing the results to generate additional RDF to map the temporary identifiers to global identifiers.

8.4. Conclusions about the Semantic Extensions

Within an overall preservation environment, semantically extended DFDL may play several unique roles. While text documents (e.g., PDF, .doc, open document, etc.) have well-understood common structural models that can be used in discovery and presentation, DFDL provides a technology by which other forms of data can be brought into this system by extracting XML and RDF representations of the non-text objects.

The semantic extensions to DFDL also provide a general-purpose mechanism for extracting metadata about relations within the data and between multiple data objects. In an archiving system this is important for preservation of logical relationships, and for generating annotations to be used for discovery and access.

In addition to understanding the semantics of specific objects and their connection with reference knowledge, it is also important to capture semantics that can only be derived from the relationship of the data within the file to other files and the larger context of work. Simple examples of this could include understanding that one can potentially infer that data was created by a given author (Dublin Core “creator”) from the fact that the file is associated with a “project” created by that person or is the outcome of a process run by that person.

9. Other Features

In the course of developing DFDL, a number of features were considered which were omitted from Version 1 of the DFDL specification.

One direction that DFDL may be extended is to support higher-level concepts for the storage of numeric data, especially in multidimensional arrays. Version 1 of the DFDL focused on data structures typically found in business data processing. These structures are useful for all kinds of data, but there are cases that are awkward to express in these structures.

Scientific data and imagery are characterized by large, multidimensional arrays of numbers. These arrays are stored in computer memory as one dimensional structures with a number of different schemes (e.g., [50]), which can be used to map the logical array to physical storage.

The DFDL could be extended to have annotations that express these schemes at the level of an array. That is the schema would define the logical layout of the array (e.g., a 10 x 50 x 5

array of floats), and a DFDL annotation would define its storage layout in a succinct description, such as:

3D array of float
row major order
missing values should be set to 0

A second area for extension would be capability to include or link with particular software modules. For example, a data structure within a file could be marked as data to be processed by a particular algorithm. In a sense, this notion extends the idea of MIME types to objects within a file or container. While this concept departs from the purely procedural model of XML/DFDL, it is a very natural use of DFDL and would benefit from tight integration with the DFDL language.

IV. Conclusion

10. Future Work

Completion of the specification has established the necessary foundation for the development of useful DFDL tools and services. The next step will be one or more implementations, including at least one open source reference implementation. These implementations will open the way to integration with tools and services, and the use in real applications.

In our work we have developed two partial implementations of DFDL, Defuddle [30, 57] and Daffodil [52]. This experience suggests that implementing a DFDL parser is feasible but not trivial. Implementation requires deep understanding of the DFDL specification [51], which, as discussed above, is quite complex. The ideal implementation staff would have experience in the design of parsers and a deep understanding of the XML Schema language [67, 69]. Even with highly expert staff, a complete implementation could require a programmer-year.

The Daffodil parser developed as part of the "Innovative Systems and Software: Applications to NARA Research Problems" project could be the basis for a complete implementation of the DFDL. As discussed above and in [17], significant work will be needed to achieve this goal. However, from what we know today, it probably would be easier to build on Daffodil than to start over.

The Defuddle and Daffodil parsers were designed to implement only the parsing step (data to XML), we did not attempt to implement the "unparser" facility which is important for some applications. From our experience, it is not clear how difficult it may be to implement unparsing. Some aspects and simple cases are straightforward, but there are cases where the XML does not contain all of the information needed to create the original data. This and similar cases will need to be worked out for any complete DFDL unparser.

Implementation of the DFDL would be greatly aided by the availability of a test suite, along with conformance standards. The DFDL working group has begun work such a test suite, but considerable effort is still required.

A parser implementation is a necessary step for all uses of DFDL. In the course of our work we have identified several next steps that would build on a successful parser. These include integration into tool and workflows, and the development of a registry for DFDL schemas.

One valuable task would be to integrate DFLD support into XML development tools, especially the Eclipse IDE. Eclipse has plugins to support XML and XML schema editing ([58]), which could be expanded to assist creation, validation, and debugging DFDL annotations. This would be a moderately difficult project (due to the complexity of Eclipse and DFDL), but would greatly simplify the use of the DFDL.

A DFDL parser is a natural addition to a preservation or data processing workflow. DFDL might be included in an iRODS microservice, as a component in a web-based service such as the ISDA tools ([3, 32]), or in a semantic content management system such as Medici [27, 45, 55].

Integration would be fairly straightforward, though DFDL has a relatively complex control and data flow. The parser has several inputs (data, schema, optional XSL) and output (XML, RDF, optional post processed data). In addition, the parser needs enough context to determine which schema to apply to a given file, and what to do with the output. Some of this context is conceptually tricky, such as managing the identity and references to the original and derived data (e.g., extracted metadata might be “about” the original, the derived XML, or both, depending on the purpose).

An important feature of DFDL is that the annotated schemas are valuable machine-readable documentation that needs to be shared. When using DFDL, a critical first step is to locate the right schema for a given dataset, which in many cases should be a shared community resource. It should be noted that there a particular dataset might have several DFDL schemas that could be used, for example, to extract different subsets from the data.

Clearly, it would be very useful to have directories of DFDL Schemas that are accessible via well-known URLs. This is a straightforward service to create and maintain and would be a natural addition to a service such as the ISDA conversion software registry [3, 31].

11. Summary

The Data Format Description Language (DFDL) is an elegant concept that has many important uses. The language is based on the XML Schema language, which provides a solid theoretical and practical foundation—at the cost of complexity and difficulty of use. The specification is complete and ready for implementation. The critical next step is

production of an open source reference implementation. NCSA's Daffodil parser is a solid beginning for such an implementation, but requires significant work to finish it.

This document has presented important use cases for DFDL, including interoperation and preservation of data. These use cases highlight the generality of DFDL: it is a technology that provides a standard, general solution to an array of problems, which can replace many domain- and system-specific solutions. Of course, DFDL does not solve all problems, as discussed above.

An implementation of the DFDL will be a general-purpose parser generator, driven by DFDL schema documents. A given DFDL parser is comparable to a custom-written parser for the same dataset. The advantages of using DFDL rather than writing code are the declarative model and the generality of the code.

The generality means that only one software stack is needed, based on a documented standard. This is critical for maintaining access to thousands of data formats. The declarative model means that the logical format of the data is explicitly documented (in machine-readable form) rather than "hidden" in code. This is a crucial advantage for both interoperability and long-term preservation of data.

The advantages of not without cost. Creating a DFDL schema is potentially complex and time-consuming. The resulting parser may well perform poorly compared to custom written code for the same case. The implications of performance issues depend on the circumstances and requirements, and it remains to be seen how significant these issues are. The difficulty of developing DFDL schemas can be partially mitigated by developing tools. Also, the effort to create a DFDL schema is partly offset by the high value of the product: the schema is more portable, reusable, and sustainable than custom written code. Finally, implementing the DFDL itself is relatively difficult. The DFDL specification is complex and requires deep understanding of XML Schema.

We have explored semantic extensions to the DFDL. Clearly, interoperation and preservation must be about maintaining the meaning of the data, not just access. Beyond "getting into XML" with DFDL, it is important to get the data "into the web" of linked data. Our two-step approach is a pragmatic first step.

Acknowledgements

The opinions expressed in this report represent the views of the author. Much of the work reported here was conducted as part of the "Innovative Systems and Software: Applications to NARA Research Problems" project, 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.

Many individuals contributed to the work discussed here, including James D. Myers, Tara Gibson, Jason Kastner, Alejandro Rodriguez, and Joe Futrelle. I thank them for their efforts. Errors in this report are the fault of the author.

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. Apache Software Foundation. *Welcome to JaxMe 2*. 2005, http://ws.apache.org/jaxme/.
3. Bajcsy, P., R. Kooper, L. Marini, K. McHenry, and M. Ondrejcek. *A Framework for Understanding File Format Conversions*, 2011. http://st13.nist.gov/workshop/papers/02_03_FileConversionServices_v2.pdf
4. Berglund, Anders, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon, *XML Path Language (XPath) 2.0*. W3C Recomendation, 2010. http://www.w3.org/TR/xpath20/
5. Berners-Lee, Tim. *An readable language for data on the Web.* 1998, http://www.w3.org/DesignIssues/Notation3.html.
6. Blackburn, Kent, Albert Lazzarini, Tom Prince, and Roy Williams, *XSIL: Extensible Scientific Interchange Language*. California Institute of Technology Technical Report CaltechCACR:CACR-1999-171], 1999. http://cacr.library.caltech.edu/29/
7. Boag, Scott, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, *XQuery 1.0: An XML Query Language*. W3C Recommendation, 2010. http://www.w3.org/TR/xquery/
8. Brickley, Dan and Libby Miller, *FOAF Vocabulary Specification 0.9*. FOAF, 2007. http://xmlns.com/foaf/spec/
9. Committee on Technology National Science and Technology Council, *THE NETWORKING AND INFORMATION TECHNOLOGY RESEARCH AND DEVELOPMENT PROGRAM*. SUPPLEMENT TO THE PRESIDENT'S BUDGET FOR FISCAL YEAR 2012, 2011. http://www.nitrd.gov/PUBS/2012supplement/FY12NITRDSupplement.pdf
10. Connolly, Dan, *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*. W3C W3C Recommendation, 2007. http://www.w3.org/TR/grddl/
11. Corubolo, Fabio, Paul Watry, Ken Arnold, Jérôme Fuselier, Gonçalo Antunes, Andreas Hundsdörfer, Hans-Ulrich Heidbrink, Anders Jansson, and Klas Moberg, *Specification*

report: Document formats, Implementations, Identification Methods specifications for data used in the test bed. SHAMAN Internal SHAMAN –WP4-D4.1, 2009.
12. Cowan, John and Richard Tobin, *XML Information Set (Second Edition)*. W3C W3C Recommendation, 2004. http://www.w3.org/TR/xml-infoset
13. DFDL-WG. *OGF Standards: Data Format Description Language (DFDL)* 2011, http://www.ogf.org/dfdl/.
14. Duce, David, *Portable Network Graphics (PNG) Specification (Second Edition)*. W3C Recommendation, 2003. http://www.w3.org/TR/PNG/
15. École Polytechnique Fédérale de Lausanne. *The Scala Programming Language*. 2010, http://www.scala-lang.org/.
16. ESW Wiki. *Grddl Implementations*. 2008, http://www.w3.org/wiki/GrddlImplementations.
17. Futrelle, Joseph and Robert E. McGrath, *Daffodil Parser: Explorations of the DFDL Standard'*. National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign 2011. https://opensource.ncsa.illinois.edu/confluence/download/attachments/9273424/DaffodilEvaluation.pdf?version=1&modificationDate=1316534292159
18. GRDDL.py. *GRDDL.py*. 2006, http://www.w3.org/2001/sw/grddl-wg/td/GRDDL.py.
19. Hayes, Patrick, *RDF Semantics*. W3C W3C Recomendataion rdf-mt, 2004. http://www.w3.org/TR/rdf-mt
20. Image Spatial Data Analysis Group. *Web Services*. 2011, http://isda.ncsa.illinois.edu/drupal/content/web-services.
21. International Organization for Standardization, *Information technology - Database languages - SQL*, in *Part 14: XML-Related Specifications (SQL/XML)*. 2008.
22. ISO, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. 2003. http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf
23. JENA. *Jena GRDDL Reader* 2011, http://jena.sourceforge.net/grddl/.
24. Joint Photographic Experts Group, *Welcome to JPEG.* 2006. http://www.jpeg.org/
25. Khan, Khalid Amin, Gulraiz Akhter, and Zulfiqar Ahmad, *OIL-Output input language for data connectivity between geoscientific software applications.* Comput. Geosci., *36 (5):687-697*, 2010.
26. Klyne, Graham and Jeremy J. Carroll, *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C 10 February 2004. http://www.w3.org/TR/rdf-concepts/
27. Marini, Luigi, Rob Kooper, Joe Futrelle, Joel Plutchak, Alan Craig, Terry McLaren, and James Myers, *Medici: A Scalable Multimedia Environment for Research (poster)*, in *Microsoft E-Science Workshop*. 2010: Berkeley.
28. McGrath, Robert E., *XML and Scientific File Formats*. 2003. http://www.ncsa.uiuc.edu/NARA/XML_and_Binary.pdf
29. McGrath, Robert E., Jason Kastner, Alejandro Rodriguez, and Jim Myers, *Experiments in Data Format Interoperation Using Defuddle*. National Center for Supercomputing Applications, Urbana, 2009. http://cet.ncsa.illinois.edu/publications/Data_Interoperation.pdf
30. McGrath, Robert E., Jason Kastner, Alejandro Rodriguez, and Jim Myers, *Towards a Semantic Preservation System*. National Center for Supercomputing Applications, Urbana, 2009. http://arxiv.org/abs/0910.3152

31. McHenry, K., M. Ondrejcek, L. Marini, R. Kooper, and P. Bajcsy. *Towards a Universal Viewer for Digital Content*, 2011.
32. 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.
33. McHenry, Kenton, Rob Kooper, and Peter Bajcsy, *Towards a Universal, Quantifiable, and Scalable File Format Converter* in *5th International IEEE eScience conference (IEEE e-Science 2009)*. 2009: Oxford. http://isda.ncsa.uiuc.edu/peter/publications/conferences/2009/eScience09FileConversion.pdf
34. Microsoft, *Microsoft Portable Executable and Common Object File Format Specification*. 2010. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v8.docx
35. Microsoft. *Export XML data*. 2011, http://office.microsoft.com/en-us/excel-help/export-xml-data-HP010206401.aspx.
36. Microsoft. *Import XML data*. 2011, http://office.microsoft.com/en-us/excel-help/import-xml-data-HP010206405.aspx.
37. Moore, Reagan, *Building Preservation Environments with Data Grid Technology.* American Archivist, *69 (1):139-158*, 2006. http://archivists.metapress.com/content/176p51l2w5278567
38. Moore, Reagan W., Richard Marciano, Arcot Rajasekar, Antoine de Torcy, Chien-Yi Hou, Leesa Brieger, Jon Crabtree, Jewel Ward, Mason Chua, Wayne Schroeder, Michael Wan, and Sheau-Yen Chen, *NITRD iRODS Demonstration*. UNC, 2009. http://irods.org/pubs/iRODS_NITRD-Report-0910.pdf
39. Moore, Regan, *Towards a Theory of Digital Preservation.* The International Journal of Digital Curation, *3 (1):63-75*, 2008. http://ijdc.net/index.php/ijdc/article/viewFile/62/41
40. Moreau, Luc, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, Jr. George Chin, Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, Susan Davidson, Ewa Deelman, Luciano Digiampietri, Ian Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole Goble, Jennifer Golbeck, Paul Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Scheidegger, Karen Schuchardt, Margo Seltzer, Yogesh L. Simmhan, Claudio Silva, Peter Slaughter, Eric Stephan, Robert Stevens, Daniele Turi, Huy Vo, Mike Wilde, Jun Zhao, and Yong Zhao, *Special Issue: The First Provenance Challenge.* Concurr. Comput. : Pract. Exper., *20 (5):409-418*, 2008. http://portal.acm.org/citation.cfm?id=1350753
41. Myers, Jim and Alan Chappell. *Binary Format Description (BFD) Language*. 2003, http://collaboratory.emsl.pnl.gov/sam/bfd/.
42. mysql.com. *12.2.7. LOAD XML Syntax*. 2011, http://dev.mysql.com/doc/refman/5.6/en/load-xml.html.
43. National Archives and Records Administration. *Advanced Research Projects: NARA's Transcontinental Persistent Archives Prototype (TPAP)*. 2011, http://www.archives.gov/ncast/tpap.html.

44. National Institute of Standards and Technology, *Database language SQL*. 1992, National Institute of Standards and Technology: Gaithersburg, MD.
45. NCSA. *Medici Content Management System*. 2010, http://medici.ncsa.illinois.edu/.
46. NCSA. *Daffodil*. 2011, https://opensource.ncsa.illinois.edu/confluence/display/DFDL/Home.
47. Odersky, Martin, Lex Spoon, and Bill Venners, *Programming in Scala*, Mountain View, Artima Developer, 2008.
48. Oracle. *Using XML in MySQL 5.1 and 6.0*. 2010, http://dev.mysql.com/tech-resources/articles/xml-in-mysql5.1-6.0.html.
49. Oracle. *12.2.7. LOAD XML Syntax*. 2011, http://dev.mysql.com/doc/refman/5.6/en/load-xml.html.
50. Pissanetzky, Sergio, *Sparse Matrix Technology*, London, Academic Press, 1984.
51. Powell, Alan W, Michael J Beckerle, and Stephen M Hanson, *Data Format Description Language (DFDL) v1.0 Specification*. Open Grid Forum, 2011. http://www.ogf.org/documents/GFD.174.pdf
52. Rodriguez, Alejandro and Robert E. McGrath, *Daffodil: A New DFDL Parser*. NCSA, 2010. http://cet.ncsa.illinois.edu/publications/Daffodil-ANewDFDLParser.pdf
53. Rodriguez, Alejandro and Robert E. McGrath, *Some Notes of comparison between DFDL and Defuddle*. NCSA, 2010. http://cet.ncsa.uiuc.edu/publications/Review_of_Defuddle.pdf
54. Rose, K. H., S. Malaika, and R. J. Schloss, *Virtual XML: a toolbox and use cases for the XML world view.* IBM Syst. J., *45 (2):411-424*, 2006.
55. Simeone, Michael, Jennifer Guiliano, Rob Kooper, and Peter Bajcsy, *Digging into data using new collaborative infrastructures supporting humanities-based computer science research.* First Monday [online], *16 (5)* 16 April 2011. http://www.firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/3372/2950
56. STEP Tools Incorporated. *ISO 10303 STEP Standards*. 2009, http://www.steptools.com/library/standard/.
57. 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.
58. The Eclipse Foundation. *Model Development Tools (MDT)*. 2011, http://www.eclipse.org/modeling/mdt/?project=xsd#xsd.
59. The HDF Group. *HDF5*. 2011, http://www.hdfgroup.org/HDF5/.
60. The National Archives. *Welcome to PRONOM*. 2011, http://www.nationalarchives.gov.uk/pronom/.
61. Unidata. *NetCDF*. 2003, http://my.unidata.ucar.edu/content/software/netcdf/index.html.
62. Unidata. *The NetCDF Markup Language (NcML)*. 2003, http://www.unidata.ucar.edu/packages/netcdf/ncml/.
63. W3C. *Extensible Markup Language (XML)*. http://www.w3.org/XML.
64. W3C. *XSL Transformations (XSLT) Version 1.1*. 2000, http://www.w3.org/TR/xslt11/.
65. W3C. *GRDDL Working Group*. 2008, http://www.w3.org/2001/sw/grddl-wg/.
66. W3C. *Resource Description Framework (RDF)*. 2010, http://www.w3c.org/RDF.

67. W3C. *XML Schema*. 2010, http://www.w3.org/XML/Schema.html.
68. W3C. *Web Ontology Language (OWL)*. 2011, http://www.w3.org/2001/sw/wiki/OWL.
69. Walmsley, Patricia, *Definitve XML Schema*, Upper Saddle River, Prentice Hall PTH, 2002.
70. Watry, Paul, *Digital Preservation Theory and Application: Transcontinental Persistent Archives Testbed Activity.* The International Journal of Digital Curation, *2 (2):41-68*, November 2007. http://www.ijdc.net/ijdc/article/view/43
71. Westhead, Martin and Mark Bull, *Representing Scientific Data on the Grid with BinX – Binary XML Description Language*. EPCC, University of Edinburgh, , 2003. http://www.epcc.ed.ac.uk/~gridserve/WP5/Binx/sci-data-with-binx.pdf
72. Westhead, Martin, Ted Wen, and Robert Carroll, *Describing Data on the Grid*, in *Fourth International Workshop on Grid Computing*. 2003: Phoenix. http://ieeexplore.ieee.org/iel5/8914/28196/01261708.pdf
73. Whiting, Mark A., Wendy Cowley, Nick Cramer, Alex Gibson, Ryan Hohimer, Ryan Scott, and Stephen Tratz. *Enabling massive scale document transformation for the semantic web: the universal parsing agent*, 2005, 23-25.