Mapping Physical Formats to Logical Models to Extract Data and Metadata: The Defuddle Parsing Engine

Tara D. Talbott[1], Karen L. Schuchardt[1], Eric G. Stephan[1], and James D. Myers[2]

[1] Pacific Northwest National Laboratory, P.O. Box 999 Richland, WA 99352, USA
{Tara.Talbott, Karen.Schuchardt, Eric.Stephan}@pnl.gov
[2] National Center for Supercomputing Applications, 1205 W. Clark St. MC-257
Urbana, IL 61801
jimmyers@ncsa.uiuc.edu

Abstract. Scientists, motivated by the desire for systems-level understanding of phenomena, increasingly need to share their results across multiple disciplines. Accomplishing this requires data to be annotated, contextualized, and readily searchable and translated into other formats. While these requirements can be addressed by custom programming or obviated by community standardization, neither approach has 'solved' the problem. In this paper, we describe a complementary approach – a general capability for articulating the format of arbitrary textual and binary data using a logical data model, expressed in XML-Schema, which can be used to provide annotation and context, extract metadata, and enable translation. This work is based on the draft specification for the Data Format Description Language and our open source "Defuddle" parser. We present an overview of the specification, detail the design of Defuddle, and discuss the benefits and challenges of this general approach to enabling discovery, sharing, and interpretation of diverse data sets.

1 Introduction

Scientists generate a wide range of data files in the course of their research. These files are generated from instruments performing measurements on physical systems, computer simulations predicting aspects of a physical system, and manually assimilated knowledge (often in spreadsheet form.) Individual file formats can vary greatly depending on the particular experimental requirements and often evolve rapidly over time. Motivated by the desire for systems-level understanding of complex phenomena, this data increasingly needs to be shared across disciplines and transformed for different analysis contexts. Beyond standard file formats, which have met with various levels of success [1-3], scientists employ strategies of custom programming and prescriptive parsers to support sharing and collaboration of their file data. Custom parsers can be effective and efficient but problems arise as the number of formats increases. Prescriptive parsers such as NetCDF [4] and HDF [5], where the data must adhere to a pre-specified, but self-describing format and structure have been successful within certain communities, but not taken hold in others. Where standards are successful, the standards tend to become legacy formats themselves over time as new methodologies or instruments are developed. Additionally, there is a push to retain raw digital data for preservation purposes [6]. In short, non-standardized and legacy file formats will continue to play a crucial role in scientific research necessitating technologies to enable sharing, discovery and transformation of these formats.

The Extensible Markup Language (XML) allows us to represent the logical structure of data elements in a file, making it available to various tools, such as databases and query languages. XML tagged data can be easily manipulated using a higher level language such as XML Stylesheet Language Translation (XSLT) and formatted for viewing on multiple devices, or translated into different formats. However, most scientific data is not currently in XML and there are often benefits to maintaining custom formats. For example, XML tagged data tends to be quite verbose and not all data types, arrays in particular, are handled well. However, extending XML technology to handle arbitrary un-tagged, binary and textual files would make the extensive XML tools applicable to scientific data and provide analogous benefits.

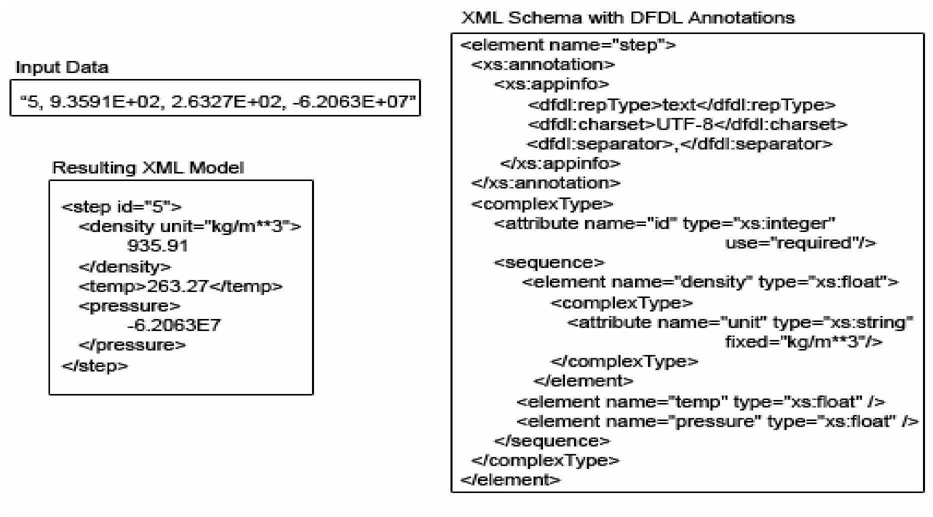
Descriptive parsers can be used to link raw physical formats into a logical model expressed as XML. With this approach, the existing data structure, the format of the data types and the mechanisms to translate it are defined in a descriptor file. A generic parser engine ingests the descriptor and the data, applies the transformation, and produces the desired result. Such a generic engine can be applied to metadata extraction as well as data transformation to greatly reduce the effort required to discover and interpret legacy data, automate transfer of data from one program to another (e.g. acquisition to analysis to visualization), and support the reuse and fusion of data across multiple domains allowing scientific communities to discover, manage, and share diverse data sets while maintaining it in its original format.

2 Background

Recently, descriptive parser approaches have received increasing attention. One effort, the Binary Format Description (BFD) language, was based on the Extensible Scientific Interchange Language (XSIL) [7], a language designed for processing scientific data, including multiple streams and arrays. The BFD parser, in conjunction with XSLT, was used by scientific computing environments for the extraction of metadata and data translation. While successful in some cases, there were many cases where BFD capabilities were not rich enough. For example, BFD was unable to map to an arbitrary XML schema, requiring an additional XSLT translation. The research from the BFD effort contributed to the production of the parser described in this paper.

The BinX descriptive parser supports the description of the content, structure and physical layout (endian-ness, blocksize…) of binary files. BinX was designed to enable transparent transfer of data between diverse platforms. However, BinX was designed to support only binary files and, as with BFD, supports limited semantics [8]. An independent, but similar effort is the Earth Science Markup Language (ESML) which is built with the intent that users can write external files to describe the structure of any earth science dataset. Applications can utilize the ESML library to parse this description file and transparently decode the data format [9]. However, the library contains several limitations; not all features, such as handling multiple wildcards, ‘if’ statements, or specific indexes of collections, are implemented, and, similar to BFD, a predefined XML model limits extensibility [10]. Another effort, designed primarily for understanding space data, the Enhanced Ada SubseT (EAST), allows users to describe a given data format and use tools to access data in that format [11]. Finally, the Universal Parsing Agent (UPA) was developed to ingest, transform, and add descriptive content markup to text data. UPA provides an accessible user interface and batch processing capabilities for handling large datasets [12]. All of these efforts have achieved success in their targeted communities but have limitations with respect to the type of data supported, extensibility, or expressiveness.

A recent development in descriptive parsers is the Data Format Description Language (DFDL) [13] specification from the Global Grid Forum. DFDL proposes to describe existing data formats, both binary and text, in a manner that makes the data accessible through generic mechanisms. DFDL is motivated by the realization that BFD, BinX, commercial tools, and domain specific efforts such as ESML, all shared a common goal and can use a common syntax while combining concepts of these languages. The specification is based on the XML Schema, which is used to define the structure and semantics of XML documents and to annotate schemas for the benefit of human readers and applications. In DFDL, XML’s extensible annotation mechanism is used to describe the data and transformations needed to populate that logical model from the input stream. The input is a sequence of bytes and the output is an XML Information Model, i.e., a set of items from the XML Information Set [14]. The transformations may require several stages (e.g., from bytes to string, then from string to integer). The DFDL specification is still under development, but is expressive enough to handle many non-trivial parsing requirements.



```
"5, 9.3591E+02, 2.6327E+02, -6.2063E+07"
```

```
<step id="5">
  <density unit="kg/m**3">
       935.91
  </density>
  <temp>263.27</temp>
  <pressure>
       -6.2063E7
  </pressure>
</step>
```

```
<element name="step">
  <xs:annotation>
     <xs:appinfo>
         <dfdl:repType>text</dfdl:repType>
         <dfdl:charset>UTF-8</dfdl:charset>
         <dfdl:separator>,</dfdl:separator>
     </xs:appinfo>
  </xs:annotation>
  <complexType>
     <attribute name="id" type="xs:integer"
                                  use="required"/>
     <sequence>
        <element name="density" type="xs:float">
           <complexType>
              <attribute name="unit" type="xs:string"
                                  fixed="kg/m**3"/>
           </complexType>
         </element>
        <element name="temp" type="xs:float" />
        <element name="pressure" type="xs:float" />
     </sequence>
  </complexType>
</element>
```

Fig. 1. Example of a DFDL schema with input data and results

For example, consider the UTF-encoded data, associated logical XML model, and sample DFDL schema shown in Figure 1. The DFDL schema is composed of an XML

Schema describing the XML model and DFDL specific annotations describing the format of the underlying data. As shown, a '*sequence*' of element definitions describes the logical XML model that will appear in the result. The '*appinfo*' annotations, known as properties, describe the format of the underlying data stream. The data type is defined as text by the '*repType*' property. The '*charset*' property defines how the incoming data stream is to be mapped to text. Finally, the '*separator*' property describes how variable length text should be read. It can be defined as a regular expression or simple text string. This example shows a very limited subset of the available annotations and properties in order to provide a perspective on the DFDL approach. Table 1 lists important capabilities of DFDL. A more detailed description of DFDL capabilities is beyond the scope of this paper.

Table 1. Key DFDL Capabilities

Support for multiple streams	Conditional logic (if, choice, any)
Basic math operations (+,-,*,/)	Looping
Pattern matching for text/binary delimiters	External transforms
Reference values within schema (for sequence length, delimiters, etc.)	
Layering (hidden elements that can be referenced, but do not display in output)	
Extensibility of basic capabilities of the DFDL parser to allow custom types and conversions	

In order to help define the components necessary in the language, several DFDL parsers are currently in development. One such implementation is the open-source Defuddle parser [15]. Defuddle supports translation and metadata extraction of arbitrary text and binary data through the use of DFDL schema descriptor files. It also optionally supports the application of style sheets to the output. The Defuddle parser is both a proof of concept of the DFDL specification and a mechanism for testing concepts which can feed back into the specification process. A specific aim of Defuddle is to demonstrate that an efficient, generic parser can be built and that such a parser can effectively address real-world examples.

3 Parser Design

Our design leverages existing tools for automatically parsing XML documents within the context of a logical model. Providing a layer of automation that makes it easy to manipulate XML encoded data in terms of a higher level logical model rather than dealing with the low level node structure directly [16]. As a result, we chose to extend a Java/XML binding compiler based on the Java Architecture for XML Binding (JAXB) specification. JAXB provides a convenient way to use XML Schema to automatically parse XML instance documents into Java classes corresponding to that schema From a design standpoint, this solution provides an off-the-shelf ingestion engine for the logical model (XML Schema), a dynamic logical model compiler (Java classes), and an XML document generator for streaming data from the classes to XML.

Figure 2 illustrates the conceptual design of the Defuddle parser. At run-time, the schema is ingested and processed to generate Java classes representing components of the logical model. These classes are then compiled using the standard Java compiler. The translation of the input data source(s) is then initiated using the JAXB XML marshaller. As the java objects are streamed by the parser, the logical model is formed by loading the required values from the data file. The XML model is streamed to an output that can then be processed by standard XML tools. The class generation process is performed automatically before translations are performed, but the compiled classes can be cached to improve performance on subsequent runs.

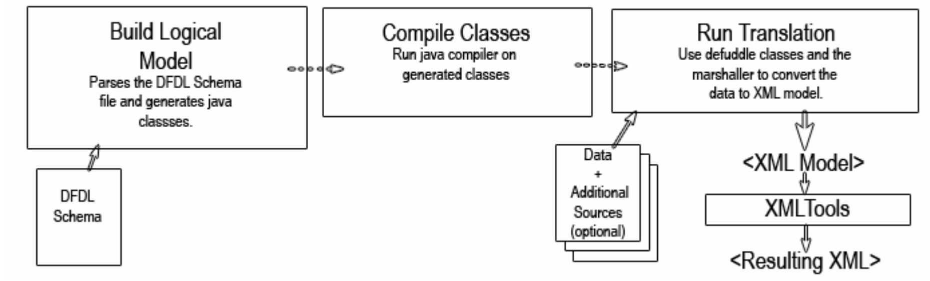
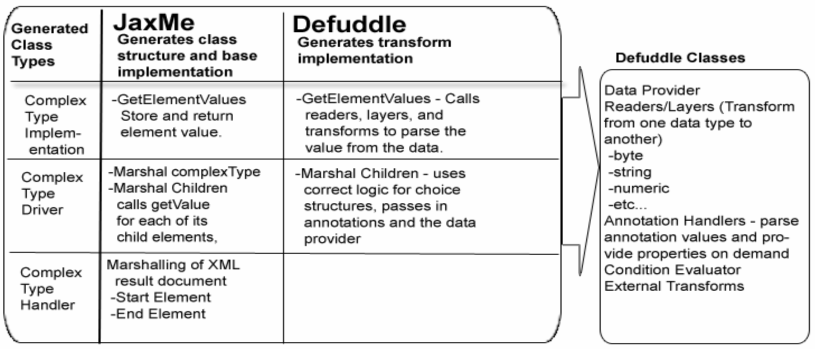

Build Logical Model
Parses the DFDL Schema file and generates java classses.
Compile Classes
Run java compiler on generated classes
Run Translation
Use defuddle classes and the marshaller to convert the data to XML model.
DFDL Schema
Data + Additional Sources (optional)
<XML Model>
XMLTools
<Resulting XML>

Fig. 2. Conceptual design of the Defuddle Parser

Defuddle is based on the Apache JaxMe project [17], an implementation of the JAXB specification. Defuddle extends the JaxMe class generator to provide the functionality needed to load data into the model and complete the transformation. Leveraging JaxMe greatly simplified the development of Defuddle. Figure 3 illustrates the types of classes generated by JaxMe and the Defuddle extensions that implement various features. Each complex type is represented by three classes: the type implementation, type driver, and type handler. Within the type implementation, values such as elements and attributes are accessed from the data stream using get<Name> methods. Vanilla JaxMe generated type implementations store and return the values, Defuddle adds content to these ‘get’ methods, which uses the annotation handlers and data provider along with other built-in Defuddle classes, such as the type readers, condition evaluators, and external transforms, to parse the required data.

While the complexType implementation classes provide information to parse individual values, the parser needs additional information to understand the structure of the data. This includes the order of the elements, the location of sequences, and the type of data to be marshaled. This functionality is found in the complexType drivers. The basic ordering and ‘get’ calls are generated by JaxMe. Defuddle extensions check for layers, hidden elements, and control sequences of unknown lengths. They also choose the correct element in conditional statements, and pass in the data provider and annotation values. The third kind of generated class is the complexType Handler; these classes are generated almost entirely by the JaxMe generator and chiefly control the marshalling of the classes to XML, ensuring the correct state when starting/closing elements and writing data.



Generated Class Types	JaxMe Generates class structure and base implementation	Defuddle Generates transform implementation
Complex Type Implem-entation	-GetElementValues Store and return element value.	-GetElementValues - Calls readers, layers, and transforms to parse the value from the data.
Complex Type Driver	-Marshal complexType -Marshal Children calls getValue for each of its child elements,	-Marshal Children - uses correct logic for choice structures, passes in annotations and the data provider
Complex Type Handler	Marshalling of XML result document -Start Element -End Element	

Defuddle Classes

Data Provider
Readers/Layers (Transform from one data type to another)
-byte
-string
-numeric
-etc...
Annotation Handlers - parse annotation values and provide properties on demand
Condition Evaluator
External Transforms

Fig. 3. Defuddle Extensions to JaxMe

Along with the JaxMe extensions, Defuddle contains additional classes to aid in parsing as shown in the right side of Figure 3. Various readers are used for converting values from one type to another, for example from byte to string, from string to multiple strings, and from string to int. Each of these readers use the annotation properties specified in the schema. Defuddle also uses a data provider to retrieve from the data stream, referencing other values in the schema, or for handling multiple input sources. When evaluating conditions, the annotations are passed on to a condition evaluator to determine the correct element to read. Defuddle also supports the calling of external transformations and integrating the results into the Defuddle transformation.

4 Discussion

Defuddle currently supports all of the features listed in Table 1. To validate the accuracy of the parser, a collection of example schemas and files have been composed. These cover a broad range of capabilities such as reading basic binary/ascii numbers, basic math operations, the ability to reference other elements within the schema, and the ability to read from multiple files or input streams. In practice, it is necessary to use a combination of these features. We have demonstrated the parser capabilities on several types of actual formats from the biological and chemical sciences including: CHEMKIN binary solution files, NWChem Molecular Dynamics property files and unstructured output files, and MicroArray and Protein-Protein Interaction Spreadsheets.

One goal of Defuddle is to demonstrate that an efficient, generic parser can be built to address real-world examples. Performance is of particular interest; can a generic descriptive parser perform as efficiently as a custom parser? A generic, pre-compiled, parser can be optimized based on the types of data and access pattern to make use of lazy parsing, avoiding unnecessary reading and caching of data. Pre-compiled schemas can also be cached to eliminate the code generation /compilation cycle. We are researching ways to better enable rapid, random access to partial data sets from tightly structured data, and to support the parsing of large data sets (through memory mapping and streaming). Additionally, with a code-generating approach, the binder can make choices when creating the schema classes to handle much of the actual parsing, such as choosing the type and length of data to be read, pre-computing the size of each data element, and requesting individual elements in tightly structured data can be served by seeking to the exact point in the data stream rather than parsing all intermediate values. For example, for a list of 100 binary floating point numbers, the location of the xth value can be computed based on the actual size of the numbers and the index of the desired number. Unfortunately this enhancement is not possible when reading varying length text, in which one must look for a separator between each element, and the size of the elements can vary based on the data being processed. Even with variable length data, intelligent parsing can still be achieved by estimating the length of the next element to read before evaluating for a delimiter, based on the size of previous elements in the sequence. If the delimiter is found before the end of the text read, the parser is able to backtrack to the position in the stream immediately after the delimiter, ensuring that no data is accidentally skipped. This estimation is often very close and speeds up parsing considerably.

When retrieving data, a more complex transformation than mere extraction may be required; this can be handled through the idea of layering. With layering, the user can describe intermediate forms of the data which are not represented in the final result. These layers are represented in the DFDL language through the use of XML Schema annotations which specify how and where each layer should be read in a stream; they also specify a name which can later be used to reference data within the layer. A layer can be accessed using annotations similar to the method used to reference other elements within the schema.

Put into practice, this type of generic parsing capability can provide a cornerstone to data sharing and collaboration environments by providing metadata extraction, translations, data slicing, and data fusion capabilities. For example, the Scientific Annotation Middleware (SAM) project provides configurable, automated metadata extraction of uploaded resources [18]. Combinations of XSLT stylesheets, Defuddle schemas, and web services are registered with SAM and run dynamically to extract metadata. For example, when a binary data file is uploaded to SAM, registered DFDL and XSLT files are accessed to generate relevant properties and store them as metadata, allowing users to automatically capture annotations. A similar mechanism can be used to provide data views - for example dynamically generated HTML pages or pages invoking Java applets for a browser-based view of the data [19]. Combined with a user environment such as the Collaboratory for Multi-scale Chemical Sciences (CMCS) [20], users can contribute data that is readily available for other users to browse, search, and access in a format suitable for their use. The availability of Defuddle is expected to reduce the number of custom translators, serve as a library of translations within applications, and provide the querying of subsets from large files. If the data description and subset queries were associated with persistent identifiers such as Life Science Identifiers (LSIDs), it should be possible to create virtual persistent identifiers for substructures and to resolve and retrieve substructures on demand [21].

5 Conclusion

In the paper, we presented a general "descriptive parser" approach to mapping physical formats to logical XML representations. This approach, based on the Data Format Description Language specification, uses data descriptions based on XML Schema extensions. Once in XML, off-the-shelf XML solutions can be applied to readily transform data, extract data and metadata, or to query the data. We detailed the design and implementation of an open-source parser engine known as Defuddle. Using real-world file formats from the chemical and biological sciences, we demonstrated that the current capabilities defined in DFDL and implemented in Defuddle are already capable of parsing a diverse set of formats. While the DFDL specification is still a work in progress, Defuddle has proved to be a useful tool in guiding specification activities and is being used to explore how extensibility can be integrated with the basic feature set.

In the future, our research will focus on extensions for internal and external transforms, layering transformations, and optimally generating data subsets from XSL translation and XPath queries. The latter feature will require smart parsing and the predetermination of the position of elements within data streams. Such features, together with the already existing capabilities, enable a range of light-weight, loosely-coupled data integration and data virtualization systems needed to support multi-disciplinary research on complex phenomena.

Acknowledgment

The research described in this paper was conducted under the Laboratory Directed Research and Development Program at the Pacific Northwest National Laboratory, a multiprogram national laboratory operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL0 1830. The authors acknowledge Robert McGrath and his work on a DFDL primer as well as helpful discussions and ongoing collaborations with members of the DFDL working group.

References

1. Critchlow, T., and Lacroix, Z., eds., Bioinformatics:Managing Scientific Data. July 2003. Morgan Kaufmann.
2. Lancashire. R, Davies, T, Spectroscopic Data: The Quest for a Universal Format, Chemistry International, Vol. 28 No. 1, January-February 2006
3. Robins, K.D., "Formatting Standards", http://www.ofcm.gov/sai/proceedings/pdf/02_panel2-3.pdf
4. netCDF Unidata: "netCDF": http://my.unidata.ucar.edu/content/software/netcdf/index.htm
5. HDF: http://hdf.ncsa.uiuc.edu/
6. Davies, T., "Cometh a Digital Dark Age?", Chemistry International Vol 24, No. 6, November 2002
7. Extensible Scientific Interchange Language: http://www.cacr.caltech.edu/SDA/xsil/
8. Binary XML Description Language: http://www.edikt.org/binx
9. Environmental Science Markup Language: http://esml.itsc.uah.edu/index.jsp

10. Environmental Science Markup Language: http://esml.itsc.uah.edu/limitations.html
11. Enhanced Ada Subset (EAST): http://east.cnes.fr/english/index.html
12. Whiting MA, WE Cowley, NO Cramer, AG Gibson, RE Hohimer, RT Scott, and SC Tratz. 2005. "Enabling Massive Scale Document Transformation for the Semantic Web: the Universal Parsing Agent." Proceedings of the 2005 ACM symposium on Document Engineering. pp 23-25. ACM Press, New York, NY
13. Data Format Description Language: http://forge.gridforum.org/projects/dfdl-wg
14. John Cowan and Richard Tobin (eds), "XML Information Set" W3C Working Draft 16 March 2001, http://www.w3.org/TR/xml-infoset .
15. Defuddle Sourceforge Project: http://sourceforge.net/projects/defuddle
16. Java Architecture for XML Binding : http://java.sun.com/webservices/jaxb
17. Apache JaxMe: http://ws.apache.org/jaxme/
18. Scientific Annotation Middleware: http://collaboratory.emsl.pnl.gov/sam/
19. Talbott TD, MR Peterson, J Schwidder, and JD Myers. 2005. "Adapting the Electronic Laboratory Notebook for the Semantic Era." 2005 International Symposium on Collaborative Technologies and Systems, pp. 136-143. IEEE Computer Soc., Los Alamitos, CA.
20. Collaboratory for Multi-Scale Chemical Science: http://cmcs.org
21. Myers, J. "Fine-grained References into Binary Data and Data Virtualization Services", Presented at W3C Workshop on Semantic Web for Life Sciences 27-28 October 2004, Cambridge, Massachusetts USA