# Daffodil Parser: Explorations of the DFDL Standard

**Joseph Futrelle**
**Robert E. McGrath**
**National Center for Supercomputing Applications**
**University of Illinois, Urbana-Champaign**
**June, 2011**

## 1. Introduction

Daffodil is a DFDL parser written almost entirely in Scala [5]. It supports much of the Open Grid Foundation Data Format Description Language (DFDL) specification [4], albeit incorrectly in some cases, and is missing some of the features required for minimal DFDL compliance.

It also supports some extended capabilities, notably GRDDL output.

Daffodil was created in response to the question, "what would it take to create a conformant DFDL parser". The resulting effort demonstrated that creating a parser is feasible, but it is not a correct implementation of the specification, although the Daffodil codebase could be used to develop an implementation of DFDL.

This document presents an assessment of the Daffodil parser. Section 2 describes the general design of a DFDL parser, and Section 3 describes the architecture of the Daffodil parser. Section 4 discusses the state of the Daffodil implementation, identifying several major problems to be addressed. Section 5 sketches steps that would be needed to address these issues. Section 6 concludes.

## 2. The Data Format Description Language (DFDL) and DFDL parser architecture

The Data Format Description Language (DFDL) Specification is a Proposed Recommendation of the Open Grid Foundation [4]. The DFDL language is expressed in a set of annotations on an XML Schema, which follow the XML Schema specification [6, 7]. A valid DFDL Schema must be a valid XML Schema, but 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.[1] 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 1).

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.

[1] 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.

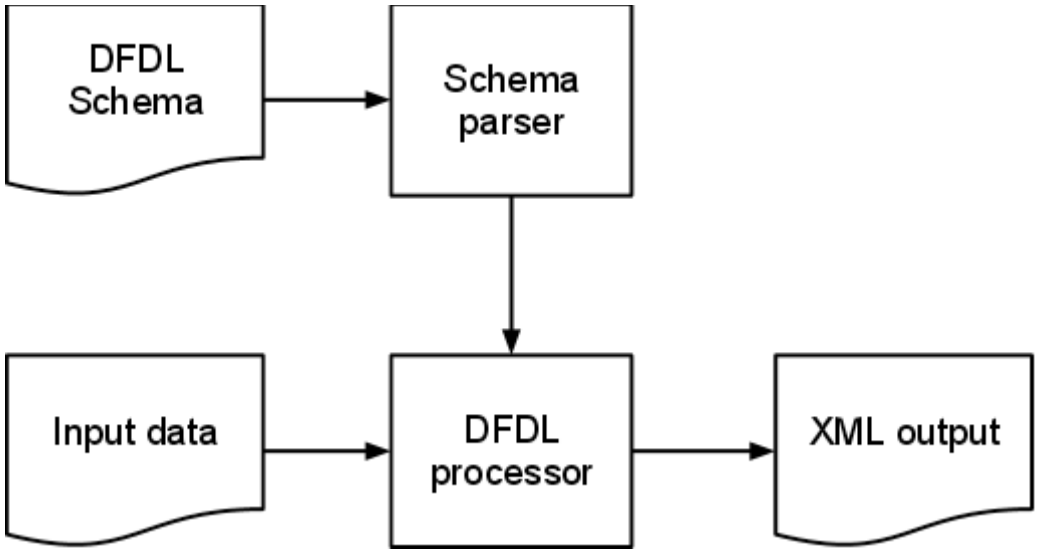


**Figure 1. DFDL Workflow**

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 (for our earlier work using of GRDDL, see [2]).

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" ([4], p.10). This latter constraint ensures that the parser does not need to support arbitrarily complex rearrangements of data. For the most part, the input data can be treated as a stream.

### 3. Daffodil architecture

The Daffodil parser is a partial implementation of concepts from the DFDL specification, written in the Scala programming language [1, 3]. Daffodil is a fresh implementation, it does not build on any previous code.

Daffodil's main program executes the overall parsing steps. The parser:

- 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.

### 3.1. Architecture components

Figure 2 shows the main classes in Daffodil.

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.

### 3.2. 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

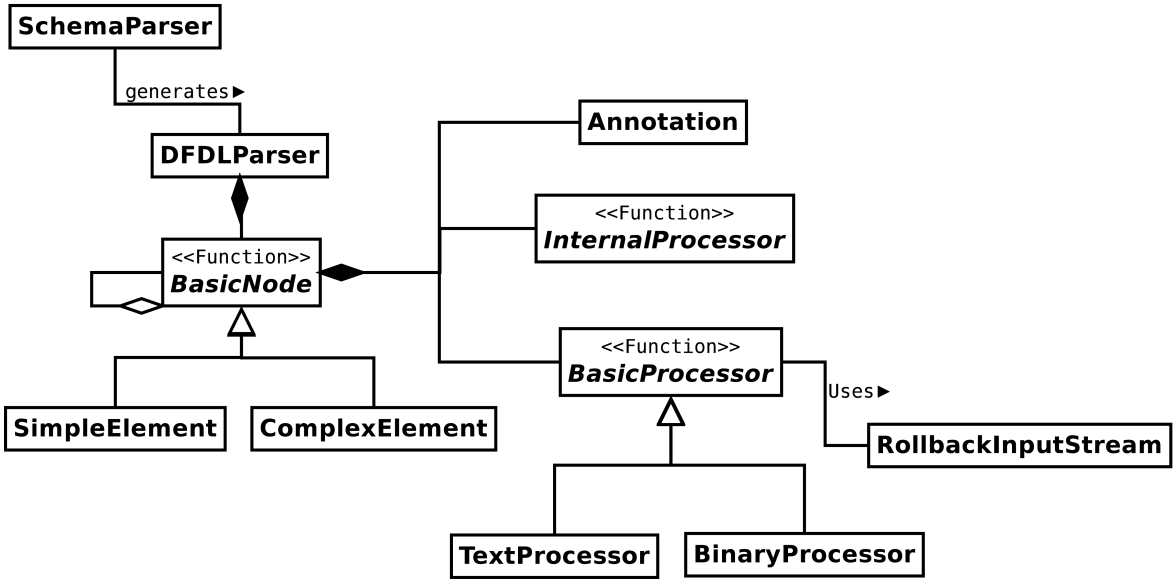


**Figure 2. Daffodil architecture. (Note that this is slightly out of date: DFDLParser is**

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.

## 4. 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 the next section suggests a plan for addressing them.

### 4.1. Schema parsing issues

The schema parsing was evaluated by code inspection and two test suites. One suite was developed with the parser ("internal"), the other was a preview of the DFDL Working Group's test suite. These investigations uncovered deep and serious problems with the Daffodil implementation.

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. This suite is under development, but we have used an early release of the tests. Daffodil does not pass any of these tests, due to issues discussed below.

In addition, the Daffodil source itself contains test cases. An immediate problem with the current implementation is that the internal test schemas do not conform to the DFDL 1.0 specification.

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 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.

### 4.2. Processing issues

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.

### 4.3. Code-level documentation

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 spec has been finalized.

## 5. Development plan

The critical problem with Daffodil's schema parser is its implementation of property scoping (i.e., defaulting). The Daffodil parser differs sufficiently from the rules specified in section 8 of the DFDL specification that it may have significant architectural implications for the schema parser. It may be necessary to rewrite the schema parser entirely, starting with the rules outlined in section 8, including the referencing rules described in section 8.3 ([4], p.49-54).

Inspection of the code shows that there are tiles (i.e., blocks of code copied in multiple places) in the code used to support Daffodil's incorrect model of scoping. We suggest that a more dynamic approach be used, in which properties are treated as keys in a dynamic data structure rather than as Scala members. This would decouple scoping rules from the code used to access the values of properties, allowing the code to be more rapidly adapted to new versions of the specification, or for experimental extensions to the specification. This change requires a significant redesign of the schema parser.

On the other hand, the text and binary processors in Daffodil appear to be largely decoupled from the schema parser. From our investigations, it appears that they can be configured and run independently, although some work might need to be done on the way they produce output by modifying DOM nodes passed to them.

Bringing the Daffodil parser up to a useful level of conformance will require additional investigation and implementation. We sketch the main tasks here.

1. Verify that text and binary processors can be configured and used programmatically (i.e., not using the schema parser), and make any necessary refactoring to ensure that they are completely decoupled from parsing logic
2. Implement section 8 of the DFDL specification using a “dynamic” model of properties (e.g., something like a map, with inheritance via a chain of explicit references), that can support defaulting and overriding for arbitrary representation properties.

3. Build a new schema parser using the mechanism developed in step 2, skipping features listed in section 21 ("Optional DFDL features") as needed.
4. Test using best available test cases, preferably developed by external partners

In doing these steps, code-level documentation should be developed that references the relevant sections of version 1.0 of the specification, so that code that implements a feature can be associated with the definition of that feature.

## 6. Summary

Daffodil is an experimental codebase that provides implementations of some of the concepts and ideas from the Data Format Definition Language specification. *It is not an 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.

*It is not recommended that the Daffodil codebase be characterized or advertised as an implementation of DFDL.* Rather, it is properly understood as a coherent, integrated set of implementation experiments demonstrating how a DFDL processor can be architected and implemented. To the (as yet undetermined) extent that some of its components can be reused, Daffodil can also be treated as a set of utilities supporting a future implementation.

## 7. Acknowledgements

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

## References

1. École Polytechnique Fédérale de Lausanne. *The Scala Programming Language*. 2010, http://www.scala-lang.org/.
2. 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
3. Odersky, Martin, Lex Spoon, and Bill Venners, *Programming in Scala*, Mountain View, Artima Developer, 2008.
4. 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
5. Rodriguez, Alejandro and Robert E. McGrath, *Daffodil: A New DFDL Parser*. NCSA, 2010. http://cet.ncsa.illinois.edu/publications/Daffodil-ANewDFDLParser.pdf
6. W3C. *XML Schema*. 2010, http://www.w3.org/XML/Schema.html.
7. Walmsley, Patricia, *Definitve XML Schema*, Upper Saddle River, Prentice Hall PTH, 2002.