Automatically Generating DTD-Specific XML Parsers

Karl Nyberg
Grebyn Corporation
P. O. Box 47
Sterling, VA 20167-0047
703-406-4161

karl@nyberg.net

ABSTRACT

This paper presents an automated approach to creating DTD-specific XML parsers in Ada. It describes a “work in progress” with discussion of current performance, limitations and future plans.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Ada, Algorithm, Performance, Standards.

General Terms

Algorithms, Performance, Experimentation.

Keywords

DTD, XML, Ada, automation, parsing.

1. INTRODUCTION

Large quantities of scientific data are published each year by NASA. These data are often accompanied by metadata files that describe the contents of individual files of the data. One example of this data is the ASTER (Advanced Spaceborne Thermal Emission and Reflection Radiometer) [1]. Each file of data (consisting of satellite imagery) in HDF (Hierarchical Data Format) [2] is accompanied by a metadata file in XML (Extensible Markup Language) [3], encoded according to a published DTD (Document Type Definition) [6] that indicates the components and types of data in the metadata file.

Each ASTER data file consists of an image of the satellite as it passes over the earth. Information on the location of the data collected as the satellite passes is contained in the metadata file. Over time, multiple images of the same location on earth are obtained. For many purposes of analysis (erosion, building patterns, deforestation, glacier movement, etc.), these images of the same location are compared over time.

A tool parses the metadata describing these images to determine which images contain data from common locations and generates a list of files according to location. The remainder of this paper describes an approach to automatically creating a DTD-specific parser and its use in selecting images from common locations for such further analysis.

2. THE ASTER PROJECT

The ASTER project is:

> an imaging instrument flying on Terra, a satellite launched in December 1999 as part of NASA's Earth Observing System. ASTER is a cooperative effort between NASA, Japan's Ministry of Economy, Trade and Industry (METI) and Japan's Earth Remote Sensing Data Analysis Center. ASTER is being used to obtain detailed maps of land surface temperature, reflectance and elevation. The three EOS platforms are part of NASA's Science Mission Directorate and the Earth-Sun System, whose goal is to observe, understand, and model the Earth system to discover how it is changing, to better predict change, and to understand the consequences for life on Earth [4].

The Terra platform operates on a “sun-synchronous” 16-day cycle, providing 233 orbit paths during this cycle. Data from the ASTER instrument is downloaded to a ground station, processed and packaged and made available at both METI and NASA web sites. Additional higher level data products are also produced from the underlying raw data. It was an interest in producing such a higher level product (a Digital Elevation Model – DEM) that lead to this effort. (Of course, in classic form, as this effort was taking shape, an equivalent higher level product was released by the ASTER project [5].)

Images from the ASTER project come paired – a data file (contained in HDF format) and a metadata file (contained in XML). Although significantly related to the underlying project associated with the ASTER data, description of the Hierarchial Data Format and development of a parser for HDF files in Ada is not described here, but rather is the subject of a separate, but related, paper [11].

3. EXTENSIBLE MARKUP LANGUAGE

Extensible Markup Language (XML) is a subset of the Standardized General Markup Language (SGML) technology for document descriptions designed for use in the web environment in a manner similar to the HyperText Markup Language (HTML). The two components of XML technology are the Document Type Definition (DTD) file, which present a description of the document and individual XML document files, which contain the data.

3.1 DTD

At the simplest level, a DTD is a definition of the structure of the components of a document. The DTD defines the elements of an associated document and the hierarchy of the elements within that document. A simple DTD example describing a person (e.g., person.dtd):

```
<!ELEMENT Person (Name, Height, Weight)>
<!ELEMENT Name (First, Middle?, Last)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Middle (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Height (#PCDATA)>
<!ELEMENT Weight (#PCDATA)>
```

The !ELEMENT keyword indicates a toplevel designator and the #PCDATA keyword indicates arbitrary text. The parenthesized grouping indicates that the contained elements appear in a hierarchial manner. The question mark after the “middle” indicates that it is optional (other punctuation is used to indicate zero or more, one or more, etc.).

3.2 XML

An XML document indicates the DTD to which it conforms and the associated contents. For example (the ellipsis below is used to indicate material irrelevant to this discussion, but necessary for a well-formed document):

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE person ...>
<Person>
        <Name>
                <First>Karl</First>
                <Last>Nyberg</Last>
        </Name>
        <Height>1.75</Height>
        <Weight>190</Weight>
</Person>
```

Notice that we have specifically mixed units of height and weight for both Metric and English units in the document. Nothing in the DTD specifies that information. Such information would have to be validated outside of the DTD.

4. USING GENERATED CODE

4.1 DATA TYPES

The DTD parser reads in the DTD file, calculates the hierarchical relationship among the elements and generates a set of data types specific to the underlying elements of the DTD. A portion of the types generated for the above DTD (all generated code is available online at [8])

```
package person is


--
-- First
--


type Element_First is record
```

```
   The_First : String_Ptr;
end record;

type Element_First_Ptr is access Element_First;

type Element_First_Ptr_Array is array (natural range <>) of Element_First_Ptr;

type Element_First_Ptr_Array_Ptr is access Element_First_Ptr_Array;

package First_List is new List
   (Element_First, Element_First_Ptr, Element_First_Ptr_Array, Element_First_Ptr_Array_Ptr);
use First_List;

type Element_First_Ptr_Array_Ptr_Record (Ct : Count_Type) is record
    The_First_Ptr_Array : Element_First_Ptr_Array_Ptr;
end record;
...

--
-- Name
--

type Element_Name is record
   The_First_Ptr_Array : Element_First_Ptr_Array_Ptr_Record (ONCE);
   The_Middle_Ptr_Array : Element_Middle_Ptr_Array_Ptr_Record (OPTIONAL);
   The_Last_Ptr_Array : Element_Last_Ptr_Array_Ptr_Record (ONCE);
end record;
...

--
-- Person
--

type Element_Person is record
   The_Name_Ptr_Array : Element_Name_Ptr_Array_Ptr_Record (ONCE);
   The_Height_Ptr_Array : Element_Height_Ptr_Array_Ptr_Record (ONCE);
   The_Weight_Ptr_Array : Element_Weight_Ptr_Array_Ptr_Record (ONCE);
end record;
...

end person;
```

These types are somewhat “heavy” or verbose because they are intended to be more general than those shown in this simple example, particularly the optional and multiple occurrence situations. What these declarations provide is essentially an infrastructure into which an instance of the document can be parsed.

4.2 PARSER

Once the infrastructure for the elements in the DTD has been created, a corresponding parser for an XML document of that structure is also generated. These procedures are essentially generated in a similar hierarchical pattern to the type declarations.

```
procedure Parse_Element_Name (E : in out Element_Name_Ptr; Quantity : Count_Type) is
begin
   read_token ("Name", xml, location);
   Parse_Element_First_Ptr_Array_Ptr_Record (E.The_First_Ptr_Array, ONCE);
   Parse_Element_Middle_Ptr_Array_Ptr_Record (E.The_Middle_Ptr_Array, OPTIONAL);
   Parse_Element_Last_Ptr_Array_Ptr_Record (E.The_Last_Ptr_Array, ONCE);
   read_token ("/Name", xml, location);
end Parse_Element_Name;


...


procedure Parse_Element_Person (E : in out Element_Person_Ptr; Quantity : Count_Type) is
begin
  read_token ("Person", xml, location);
  Parse_Element_Name_Ptr_Array_Ptr_Record (E.The_Name_Ptr_Array, ONCE);
   Parse_Element_Height_Ptr_Array_Ptr_Record (E.The_Height_Ptr_Array, ONCE);
   Parse_Element_Weight_Ptr_Array_Ptr_Record (E.The_Weight_Ptr_Array, ONCE);
   read_token ("/Person", xml, location);
end Parse_Element_Person;
```

4.3 DISPLAY ROUTINES

Similar to the parsing routines, a collection of display routines are generated to walk an object hierarchy and display the contents in human-readable format. Some examples:

```
procedure Display_Element_Person (E : Element_Person) is
begin
   Display_Element_Name_Ptr_Array_Ptr_Record (E.The_Name_Ptr_Array);
   Display_Element_Height_Ptr_Array_Ptr_Record (E.The_Height_Ptr_Array);
   Display_Element_Weight_Ptr_Array_Ptr_Record (E.The_Weight_Ptr_Array);
end Display_Element_Person;

procedure Display_Element_Person_Ptr_Array_Ptr_Record
   (E: Element_Person_Ptr_Array_Ptr_Record) is
begin
   if E.The_Person_Ptr_Array = null then
      return;
   end if;
   put ("<Person>");
   new_line;
   for I in E.The_Person_Ptr_Array'range loop
      Display_Element_Person (E.The_Person_Ptr_Array (I).all);
   end loop;
   put ("</Person>");
   new_line;
end Display_Element_Person_Ptr_Array_Ptr_Record;
```

4.4 APPLICATION CODE

One of the advantages of this approach is the ability to directly reference components of an XML document using standard Ada language constructs (record structures and arrays), rather than having necessarily to hierarchically or sequentially search through the object by looking for keyword and value pairs (it will, however still be necessary to make conditional decisions based on optional or multiple occurrence components). Parsing a list of filenames containing XML documents is straightforward.

```
 for I in 1 .. Argument_Count loop
      Open (F, In_File, Argument (I));
      Docs (I) := new Person.
         Element_Person_Ptr_Array_Ptr_Record (Once);
      Parse_File (F, Docs (I).all);
      Close (F);
 end loop;
```

The following two examples show how the structure of the data types would be used for a required ELEMENT (“weight”) and for an optional ELEMENT (“Middle”) where individual XML files have been parsed into corresponding elements of the array “docs”.

```
--
-- find all the people with weight > 100
--

Put_Line ("============================");
Put_Line ("people with the weight > 100");
Put_Line ("============================");

declare
   Weight : Integer;
   Dummy : Natural;
begin
   for I in 1 .. Argument_Count loop
      My_Integer_Io.Get (Docs (I).The_Person_Ptr_Array (1).
         The_Weight_Ptr_Array.The_Weight_Ptr_Array (1).
         The_Weight.all, Weight, Dummy);
      if Weight > 100 then
         Display_Element_Person_Ptr_Array_Ptr_Record (Docs (I).all);
      end if;
   end loop;
end;

--
-- find all the people with a middle name
--

Put_Line ("=========================");
Put_Line ("people with a middle name");
Put_Line ("=========================");

for I in 1 .. Argument_Count loop
   if Docs (I).The_Person_Ptr_Array (1).
     The_Name_Ptr_Array.The_Name_Ptr_Array (1).
     The_Middle_Ptr_Array.The_Middle_Ptr_Array /= null then
        Display_Element_Person_Ptr_Array_Ptr_Record (Docs (I).all);
   end if;
end loop;
```

5. RESULTS AND LIMITATIONS

The parser generator was only implemented for the subset of DTD and XML that was encountered by the problem at hand. Additional work would be necessary to extend it to handle attribute lists (ATTLIST) and namespaces. There is not a lot of error-checking performed either, under the reasonable assumption that the XML files were well-formed when received. This assumption held for the almost 50,000 XML files that were processed.

No consideration is provided for parsing XML content using attributes, as none of the input files required this form.

A set of routines for walking an object hierarchy and deallocating the memory associated with the object needs to be generated.

The DTD used for the image data in the ASTER project was approximately 200 lines long, consisting of over 100 ELEMENTS [7]. The generated data types were about 5,000 lines of Ada and the parser an additional 4,500 lines. The display package that walked instances of the structure and displayed the contents was another 3,500 lines. Parsing the DTD and generation of the DTD-specific parse code takes mere seconds to perform. Compilation of the simple application used at this juncture is less than 30 seconds.

In a couple instances the DTD being used contained an element consisting of multiple common sub-elements, such as:

<!ELEMENT Boundary (Point, Point, Point, Point*)>

In such cases, the underlying record data structure contained components disambiguated by numeric instantiation:

```
Type Element_Boundary is record
   The_Point_Ptr_Array : ...
   The_Point_2_Ptr_Array : ...
   The_Point_3_Ptr_Array : ...
   The_Point_4_Ptr_Array : ...
end record;
```

The generated types and referencing structure is a bit verbose and clunky, to say the least at this stage, and could probably be cleaned up.

6. RELATED WORK

6.1 Existing XML Ada Parsers

There are a number of XML parsers for Ada, both native and as bindings to underlying libraries in other languages [10]. For the most part, these tools provided less DTD-specific opportunities for individual applications. This tool provides an approach similar to that with the Data Object Model (DOM) approach, but carries the approach one level further by making the references extremely concrete rather than needing to search through the DOM object (essentially a parse tree) at runtime for individual elements.

6.2 DTD Parsers and Tools

There are numerous tools for parsing DTDs and doing consistency checking and such with the underlying data. An interesting tool for hierarchical display of DTD schemas is the Matra tool - An XML DTD Parser Utility [9]. Adding a similar feature to the current software would simply require a straightforward top-down rewalking of the parsed hierarchy. Other applications could include normalization tools, cross referencers, etc.

7. PERFORMANCE

Image data from the NASA web site was downloaded to use as input to the DEM generation process. (The ASTER project subsequently generated a Global DEM using data that was derived from 1.3 million images and required over a year's time to process and create). For this effort, approximately 50,000 images and associated XML metadata files, restricted to US territories, were used.

Pundits have often complained about Ada's performance, so we were curious to see how quickly the generated code would be able to parse XML files. (Based on the adage "it's easier to make a working program fast than it is to make a fast program work", we wanted to see how well the straightforward approach performed.) As of the time of the writing of this article, there were approximately 50,000 XML metadata files, each consisting of approximately 250 lines and 10,000 characters (just over 500MB total). Reading in and parsing these files took about one minute. Cycling through all possible longitudes (-180 .. 179) and Northern latitudes (0 .. 89) to determine the set of files for each one degree tile and generate approximately 10MB of listings in 2500 files took an additional minute. All times are reported on a several year old 3.2GHz Pentium IV Dell running gcc 4.2.1 on Fedora 5 Linux with Raptor disk drives. With acceptable performance like that, there seemed no need to focus on additional speed improvements.

(We note that although the effort reported in this paper is not comparable to that reported by the ASTER project in the creation of their Global DEM. They reported that its generation took over a year in processing time to complete. Based upon their estimate of 1.3 million data files, and a linear scaling of the performance of the underlying software used in this paper, determination of the set of files for each longitude / latitude tile portion of a similar generation would take approximately a couple hours time. The processing effort required for parsing the HDF files and the actual DEM generation is another work in progress.)

8. ACKNOWLEDGMENTS

Thanks to Richard Biby for the suggestion to do something interesting with the ASTER data and to David Emery for encouraging the writeup of this approach and reviewing the rough drafts of it.

9. REFERENCES

[1] http://asterweb.jpl.nasa.gov/

[2] http://www.hdfgroup.org/products/hdf4/

[3] http://www.w3.org/TR/REC-xml/

[4] http://www.science.aster.ersdac.or.jp/en/documnts/pdf/ASTER_Ref_V1.pdf

[5] http://asterweb.jpl.nasa.gov/gdem.asp

[6] http://en.wikipedia.org/wiki/Document_type_definition

[7] http://observer.gsfc.nasa.gov/ECSInfo/ecsmetadata/dtds/DPL/ECS/ScienceGranuleMetadata.dtd

[8] http://karl.nyberg.net/Automatically_Generating_DTD_Specific_XML_Parsers/

[9] http://matra.sourceforge.net/

[10] http://www.adapower.com/index.php?Command=Class&ClassID=AdaWeb&Title=Ada+Web+and+XML

[11] Nyberg, Karl A. "Nyberg, Karl A. "Parsing Hierarchical Data Format (HDF) Files"; *SIGAda Ada Letters, to appear*...