Have a look at XML files

Arjen Markus[1]
WL | Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

About XML

Software engineering seems to be ridden with hypes: someone launches a "new" idea, a lot of noise is made over that in the various journals which leads the casual reader or listener to believe that all previous ideas are very wrong or at least completely ancient and therefore useless. The new idea inspires the creation or resurrection of a bunch of TLAs and FLAs [2] and other buzzwords. But after a while the hype is gone and the new idea is either forgotten or becomes an old and more or less well-understood tool - one of many and not one instead of many.

A couple of years ago it was the turn for *XML* - extensible markup language. In itself nothing new - a scaled-down version of SGML and a more systematic kin of HTML. Nevertheless it was seen as a "silver bullet": its magic could solve all communication problems between computer programs because data in XML format can be understood by any XML parser. The hype led to such abbreviations as DTD, XSLT, DOM and SAX, XPath and many others. It also led to numerous wel-known libraries in C, and especially C++ and Java to read and write these files. None appeared for use in Fortran programs, which meant this "technology" was not easily accessible for Fortran programmers.

This paper aims to clarify the concept of XML and to describe how it can be used in a Fortran environment. As there are at least three (public) implementations of XML parsers in Fortran, many practical obstacles have gone. Also the hype around XML seems to have settled: it is now possible to discuss its benefits and limitations in a rational manner.[3]

What does XML look like?

First and foremost, XML is a file format that can be characterised as hierarchical. Here is a small example (adapted from the documentation of an *open source* project, PLplot):

```
<?xml version="1.0"?>
<chapter id="BIBLIOGRAPHY">
  <title>Bibliography</title>
<para>
These articles are descriptions of …
</para>

<bibliography><title>References</title>
<bibliomixed>
  <bibliomset relation="article">
      <surname>Someone</surname> <firstname>A.</firstname>,
      <title role="article">Just a matter of common sense</title>,
  </bibliomset>
  <bibliomset relation="journal">
      <title>My journal</title>,
```

[1] E-mail address: arjen.markus@wldelft.nl
[2] Three-letter and four-letter abbreviations
[3] I meant the somewhat sarcastic tone of the introduction not for XML itself, but for the unbounded enthousiasm and lack of realism that has been part of many a discussion and presentation on this subject.

```
      <pubdate>1996 December</pubdate>
  </bibliomset>
</bibliomixed>

<bibliomixed>
  <bibliomset relation="article">
      <surname>Nobody</surname> <firstname>B.</firstname>,
<title role="article">Nothing new under the sun
</title>,
  </bibliomset>
  <bibliomset relation="journal">
      <title>My journal</title>,
      <volumenum>6</volumenum>,
      <pagenums>1</pagenums>,
      <pubdate>1999</pubdate>
  </bibliomset>
</bibliomixed>

</bibliography>
</chapter>
```

As you can see from this example:

- The information in an XML file consists of opening "tags" (text between < and >) and matching closing "tags" (these start with "</")
- In between the opening and closing tags, you can have character strings but also other tags. This nesting can go on indefinitely, making up a hierarchy of tags and their associated character strings.
- Opening tags may hold attributes, such the *id* for the *chapter* tag.

Because the format of the tags is fixed, you can use a single set of routines to split the data in the XML file into tags, attributes and raw strings, *without knowing the meaning or ordering of the items.* This is the simple magic of XML files.

The full standard for XML files (W3C, 2003[4]) is more extensive and defines ways of linking an XML file to external documents that describe the desired structure (so that validation is possible - this is the role of DTDs, *document type definitions*) or to set up formatting methods (XSLT: extensible stylesheet transformations) which "automatically" convert tags and their attributes into other types of files, such as HTML files for viewing the contents in an Internet browser.

For this paper, however, it is more important to see how XML can be used to read input into a program, as that is the basic requirement for any application.

Two methods: SAX and DOM

There are roughly speaking two methods to read an XML file:

- The event-based method (also known as SAX - *simple API for XML*)
- The tree-based method (or DOM - *document object model*)

With the first method, exemplified for instance by the most famous XML parser, Expat by James Clark (Clark, 2003) you read an XML file or a part of such a file, pass that on to the parser and via a *callback mechanism* your own routines are called at particular points in the parsing process so that your program can handle the information. For instance:

- one routine may be called when a new opening tag is encountered
- another is called when raw data are found

[4] A handy overview is available - Eckstein and Casablanca, 2001

These routines have to be registered before the parser can do its work. Usually you can register routines for many different "events", but the mechanism means that you have no control over the reading, like you would have with an "ordinary" file.

The second method is based on the idea that the contents of an XML file is in fact structured as a "tree" data structure (Aho et al. 1987). So, libraries of this type simply read in the whole file and store the information in some generic tree structure. This makes the reading process itself very succinct.

Once you have read the information into this data structure, the difficulty begins, however: you will have to extract the right values for whatever tasks you have in your program - there is nothing magical left. You need to give a proper meaning to all these tags yourself.

In some industries standard *electronic data sheets* are used and these provide the semantics that is otherwise lacking. The tags have a predefined meaning, the attributes they can have are limited and the whole structure of the tree is prescribed. If this is the case, then a *validating parser* can automatically check if the structure of the XML file is in accordance with this standard. It may also be possible to find ready-made "event handling" routines to transfer the data in an XML file to the program proper.

In other areas no such standard exists and then you will need to program the data retrieval yourself. Whether a SAX or DOM approach is best, depends partly on the size of the files you need to handle:

- If the file's contents can be loaded into memory, then there is no need to extract all information at once. You can use the tree structure as a hierarchical data base, to be searched when there is a need for such data.
- If it is known in advance which data items are required, you can simply scan the file once and extract all the data you need during that one pass. There would be no need to keep the contents around after that.

Limitations to the XML concept

What does not become clear in most publications on XML is that it presumes any structure in the data is caught by tags and attributes. This makes XML rather awkward to use for the storage (or transfer) of large multidimensional arrays. Here is a simple example:

Suppose a program must read a two-dimensional array of 10 by 20 numbers (that is: 20 groups of 10 numbers each). The first thing that must be agreed upon is how to define the dimensions. In C for instance the leftmost dimension would be 20, the rightmost would be 10:

C: `float array[20][10] ;`
Fortran: `real, dimension(10,20) :: array`

One possibility to capture this in an XML format is:

```
<array rows="20" columns="10">
      …. 200 numbers, separated by a space …
</array>
```

where the attributes *rows* and *columns* must be understood as giving the number of groups and the number of data in each group respectively. An important drawback is that the numbers themselves appear as an unstructured (in the sense of XML) long string of characters: every program will have to provide its own parser for splitting up these numbers - something that goes against the XML philosophy.

An alternative that is a bit lengthier but makes the grouping clearer is:

```
<array rows="20" columns="10">
   <row>
       …. first group 10 numbers, separated by a space …
   </row>
   <row>
       …. second group 10 numbers, separated by a space …
   </row>
   … another 18 rows
</array>
```

where the attributes *rows* and *columns* can be used to check that the number of rows and the number of columns is in accordance with the actual data.

This structure would probably give less trouble understanding, but it still requires the reading program to interpret the strings of characters as an array of numbers.

The last possibility I want to present here solves even that, at the cost of becoming very lengthy indeed:

```
<array rows="20" columns="10">
   <row>
       <column>… first number …</column>
       <column>… second number …</column>
       … remaining 8 columns …
   </row>
   <row>
       <column>… first number …</column>
       <column>… second number …</column>
       … remaining 8 columns …
   </row>
   …another 18 blocks like this …
</array>
```

So, the price you pay for storing large amounts of "homogeneous" data in an XML file is that you must provide either a special parser for the character strings that contain the relevant information or that you must provide additional structure.

Here is another drawback: XML files are, despite tools like XML editors, more difficult to read by human beings than a plain text file. The tags get in the way and the whole file is rather verbose. XML has simply not been designed for this purpose.

What XML *is* good at, is the storage of highly structured data, such as the bibliographical references from the first example, or meta data about a file containing the results from a large computation (which version of the program was used, which input data were used, who did the computation and when).

Reading XML in a Fortran program

Some time ago I wrote a small library to prove that it is possible to read XML files using Fortran as the implementation language (Markus, 2003). This library uses a SAX-like approach but the difference is that you do not register callback routines. Instead the main reading routine reads up to the next tag and stores the data (attributes and character data between tags) in the variables supplied by the calling program. So the approach is much more classic:

```
program readfile

   use xmlparse
```

```
   character(len=60) :: title
   character(len=20) :: fname
   character(len=20) :: structname
   logical           :: mustread
   type(XML_PARSE)   :: info

   character(len=80)                       :: tag
   logical                                 :: endtag
   character(len=80), dimension(1:2,1:20) :: attribs
   integer                                 :: no_attribs
   character(len=200), dimension(1:100)   :: data
   integer                                 :: no_data
   integer                                 :: i

   … get name of XML file (store in fname)

   mustread = .true.
   call xml_open( info, fname, mustread )

   !
   ! Check for errors
   !
   if ( xml_error(info) ) then
      … handle the errors
   else
      !
      ! Start reading the file
      !
      call xml_options( info, ignore_whitespace = ign_whitespace )

      do
         call xml_get( info, tag, endtag, attribs, no_attribs, data, &
                       no_data )
         if ( xml_error(info) ) then
            … handle the errors
         endif

         ! Just write out the tag, its attributes and the character
         !data we found

         write( 20,* ) tag, endtag
         do i = 1,no_attribs
            write( 20,*) i, '>', attribs(1,i), '<=', trim(attribs(2,i))
         enddo
         write( 20,*) (i, '>',trim(data(i)), '<', i=1,no_data)
         if ( .not. xml_ok(info) ) exit
      enddo
   endif

   call xml_close( info )

   stop
end program
```

With this library, reading an XML file means you can use the same programming techniques as with ordinary files:

- Read a piece of the file
- Identify which actions to take (trivial in the above case, but otherwise you could branch on the tag name)
- Take these actions
- Read the next piece until the end of the file

Drawbacks of this approach are:

- You have to supply enough string arrays (dynamic allocation is possible in principle, but it has not been implemented in this library yet)
- Lines in the file are assumed to be some maximum length (again, this limitation can be overcome, but has not been considered so far)

The advantage is, however that the programmer can continue to use the very familiar pattern outlined above. Still, this may be unsatisfactory: the reading process stays intermingled with the processing of the data and you only have access to the data already read. The solution is of course to use a *tree* data structure. That way you can separate the two parts:

- In a loop you read the full XML file and store its contents in the tree - this is independent of the interpretation of the data.
- After that you retrieve the relevant data from the tree structure whenever this is convenient. In fact, you do not need to deal with the fact that the data came from an XML file anymore - you can instead use this technique with other data sources as well.

Here is an excerpt of one implementation of a general tree library in Fortran (the full code is a bit lengthy, but it is part of my XML library):

The library defines the data structures that make the tree:

```
type TREE_DATA
   character(len=1), dimension(:), pointer         :: node_name
   character(len=1), dimension(:), pointer         :: node_data
   character(len=1), dimension(:), pointer         :: node_data_type
   type(TREE_DATA_PTR), dimension(:), pointer :: child_nodes
end type

type TREE_DATA_PTR
   type(TREE_DATA), pointer               :: node_ptr
end type
```

Each node has a *name*, it can have a *value* and it holds the *type* of the value (as a string).

New nodes (branches) to any existing node in the tree or to the tree itself are created with a subroutine like this (a node is found by its *name*):

```
subroutine tree_create_node( tree, name, node )
   character(len=*), intent(in)    :: name
   type(TREE_DATA), pointer        :: tree
   type(TREE_DATA), pointer        :: node

   type(TREE_DATA_PTR), dimension(:), pointer :: children

   integer                         :: error
   integer                         :: newsize

   !
   ! Create a new node, store it in the array of child nodes
   ! for this (sub)tree
   !
   call tree_create( name, node )

   if ( associated( node ) ) then
      newsize = 1
      if ( associated( tree%child_nodes ) ) then
         newsize = 1 + size( tree%child_nodes )
      endif

      … increase the array of "child" nodes and keep
      … the original information
```

```
      tree%child_nodes => children
      tree%child_nodes(newsize)%node_ptr => node
      …
   endif
end subroutine tree_create_node
```

Most of the code has to do with checking for possible errors and maintaining the information that is already present.

The tricky part is that Fortran (90/95) does not have a "polymorphic" data type. Storing character strings of different lengths, as well as arrays of integers, as well as any user-defined derived type in the same *generic* structure would seem impossible. Well, there is a trick: the function *transfer()* can mold arbitrary data into any type. So to store the data in a node, use this routine:

```
subroutine tree_put_data( tree, data, data_type, success )
    type(TREE_DATA), pointer        :: tree
    character(len=1), dimension(:)  :: data
    character(len=*), optional      :: data_type
    logical, intent(out), optional  :: success

    integer                         :: error

    … clean up any existing data

    allocate( tree%node_data(1:size(data)), stat = error )
    if ( error .eq. 0 ) then
       tree%node_data = data
       allocate( tree%node_data_type(1:len_trim(data_type)), &
          stat = error )
       if ( error .eq. 0 ) then
          tree%node_data_type = transfer( data_type, tree%node_data_type )
       endif
    endif

    …

end subroutine tree_put_data
```

and call it like this:

```
   use TREE_STRUCTURES
   type(TREE_DATA), pointer   :: tree
   type(TREE_DATA), pointer   :: node

   integer, dimension(1:100)  :: array

   call tree_create( "TREE", tree )

   …

   ! Create a new node and use the variable "node_value" (defined in
   ! the module) to transfer the data as the proper type

   call tree_create_node( tree, "NODE A", node )


   call tree_put_data( node, transfer(array,node_value), "INTEGER ARRAY" )
```

For specific types of data it is best to create small wrapper routines, so that the user is not bothered with the use of the transfer function.

With this additional module, you can now store the contents of an XML file for later use. The translation of a tag with its attributes and its character data to a tree with nodes that hold only one piece of information can be done as follows:

The fragment

```
<array rows="20" columns="10">
        <row> 1 2 3 4 5 6 7 8 9 10</row>
        <row> … </row>
        …
</array>
```

becomes a tree like:

```
"array", type="string", value=""
        has subnodes:
                "rows", type="attribute", value="20"
                "columns", type="attribute", value="10"
                "row", type="string", value=" 1 2 3 4 5 6 7 8 9 10"
                "row", type="string", value=" … "
                etc.
```

Note that the subnodes do not need to have a unique name. This makes retrieving the correct node a bit more involved. The alternative is to create unique names “on the fly”, but the drawback is that the tags in the XML file and the tags in the program no longer match exactly.

Available libraries

Like I said in the introduction, there are at this moment at least three public implementations of XML parsers in Fortran. If you do not mind interfacing to, for instance, a C library, even more options become available. The advantage of the latter approach is that you can choose a very mature and venerable library, such as Expat.

The Fortran libraries that I am aware of are:

- My own, as I have just described:
  It is a small library with some important limitations, but it allows both a DOM approach and a SAX-like method. There is no support for validation.
- A rather complete implementation by Mart Rentmeester (2003):
  This particular library has support for different character encodings (so that non-western alphabets can be used) and comes with a varying-length string module. The reading method is SAX, where one user-defined routine handles all events.
- The third implementation, by Alberto Garcia (2003), can be used not only in the context of Fortran but also in the context of the subset F. This can be an attractive feature, as F is a carefully designed sublanguage aimed at *safe* programming techniques. It offers both a SAX approach and a DOM approach.

Of course, an article like this can not provide an answer to important questions like: "should we use XML files from now on?" Such questions need to be considered carefully - beware of hypes! But there is no longer any reason *not* to consider the combination XML and Fortran.

References

Aho, Hopcroft and Ullman (1987)
Data Structures and Algorithms
Addison-Wesley, 1987

James Clark (2003)
Expat, XML library
http://expat.sourceforge.net

Robert Eckstein and Michel Casablanca (2001)
XML, pocket reference
O'Reilly & Associates, second edition, april 2001

Alberto Garcia (2003)
xmlf library
http://lcdx00.wm.lc.ehu.es/ag/xml/

Arjen Markus (2003)
xmlparse, library for reading XML files in Fortran
http://xml-fortran.sourceforge.net

Mart Rentmeester (2003)
xml_parser library
http://nn-online.org/code/xml

W3C (2003)
Official standard for XML
http://www.w3c.org/XML