Documentation:
XMLLIB - parsing XML in Fortran

Thomas Jonsson

2018-09-23

Contents

Chapter 1

What is XMLLIB?

XMLLIB is a library for parsing xml-files in Fortran. This means that it provides subroutines for extracing values from an xml file.

XMLLIB provides three different interfaces for parsing data, here referred to as the **Konz**, **Xpath** and **xml2eg** interfaces. The Konz interface is sometimes also referred to as the Classic interface. The xml2eg interface is the most flexible of the three, the most robust and also the most simple to use. The Konz and Xpath interfaces are kept only for legacy usage. The details of the interfaces are described in section 3.

Note that XMLLIB cannot be used to parse any xml file. The xml restrictions of the different interfaces are described in section 2.

The XMLLIB source is stored in the ITER git repository. To clone this repository:

```
git clone ssh://git@git.iter.org/lib/xmllib.git
```

In this repository there are examples for each of the XMLLIB interfaces.

- For the xml2eg interface, see `examples/xml2eg/actor_example.f90` and `examples/xml2eg/sample.f90` (see also chapter 6).
- For the Xpath interface, see `examples/xpath/test_xmllib_pathquery.f90`.
- For the Konz interface, see `examples/classic/sample.f90`.

Chapter 2

XML formats used in XMLLIB

There are a number of restrictions on the xml files that XMLLIB can parse. In particular, XMLLIB cannot parse:

1. Any type of attributes. As an example, XMLLIB cannot parse `<species mass="2" charge="1"/>`. This information can instead be represented using

```
<species>
  <mass>   2 </mass>
  <charge> 1 </charge>
</species>
```

2. XMLLIB cannot parse arrays of elements. Example:

```
<family>
   <person><name> Bob  </name></person>
   <person><name> Nick </name></person>
 </family>
```

cannot be parsed. Instead the two persons may be described as an array of names:

```
<family><names> Bob , Nick </names></family>
```

2.1 Restrictions in the xml format of the xml2eg interface

Using the wml2eg interface the root element has to be `/parameters`, i.e. only data under `/parameters` can be accessed using the xml2eg interface.

2.2 Restrictions in the format of the xml-schema for Konz and Xpath interfaces

The Xpath and Konz interfaces have restrictions on how the xml-schema file should be written. The main restriction is that any child-xml-element has to be specified using a reference, `ref`, to a different element on the root level.
Example: Below the element `parameter` has an explicitely declared child `node`, thus it cannot be parsed with the Konz or Xpath interfaces.

```
<xs:element name="parameters">
  <xs:complexType>
    <xs:all>
      <xs:element name="node" type="xs:float"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

By moving the declaration outside the parameter element it can be parsed.

```
<xs:element name="parameters">
  <xs:complexType>
    <xs:all>
      <xs:element ref="node" minOccurs="1"/>
    </xs:all>
  </xs:complexType>
</xs:element>
<xs:element name="node" type="xs:float"/>
```

One consequence of this limitation is that one cannot use the same name for two fields in different branches of the xml-tree unless both fields have identical format. As an example, there is no way to parse anything similar to the xml-tree below without renaming one of the elements called `node`.

```
<xs:element name="parameters">
  <xs:complexType>
    <xs:all>
      <xs:element name="integer">
        <xs:all>
          <xs:element name="node" type="xs:integer"/>
        </xs:all>
      </xs:element>
      <xs:element name="float">
        <xs:all>
          <xs:element name="node" type="xs:float"/>
        </xs:all>
      </xs:complexType>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Chapter 3

Interfaces

The interfaces provided in XMLLIB includes both subroutines to read input files to a buffer and to parse these, i.e. extract values from the data.

3.1 Reading input files

The XMLLIB was originally built for EFDA-ITM (later EUROfusion/WPCD) datastructures called CPOs and later adapted for the ITER/IMAS datastructures, called IDSs. In both these dataformats a string is referred to as an array of 132-bit characters, i.e. `character(len=132), pointer, dimension(:)`. Consequently the XMLLIB represent the xml the same way, as an CPO or IDS string.

There are two interfaces for reading xml input files. The first one reads a single file into a buffer:

```
use f90_file_reader, only: file2buffer
character(len=132), pointer :: buffer(:) => NULL()
integer :: io_unit = 1
call file2buffer('data.xml', io_unit, buffer)
```

The second interface read three files, an xml file, a schema file and a default or reference xml file, into three buffers, `param_xml`, `param_xsd` and `param_default`.

```
use xml_file_reader, only: fill_param
character(132), pointer :: param_xml(:)     => NULL()
character(132), pointer :: param_xsd(:)     => NULL()
character(132), pointer :: param_default(:) => NULL()
call fill_param( param_xml , param_xsd , param_default , &
    'input.xml' , 'input.xsd' , 'input_default.xml' )
```

The second design of the second interface is motivated by the CPO and IDS data structures includes three files as described above. As an example the IDS derived type `ids_parameters_input` includes the three strings `parameters_value`, `schema` and `parameters_default`. The

```
type(ids_parameters_input), intent(in) :: codeparam
call file2buffer('input.xml', 'input.xsd', 'input_default.xml', &
    codeparam%parameters_value, codeparam%schema, &
    codeparam%parameters_default)
```

3.2 The xml2eg interface

To parse data using the xml2eg interface one first translate the xml data into an abstract document of type `type_xml2eg_document` using the subroutine `xml2eg_parse_memory`. Once the data is read

into the document data fields can be accessed using the subroutine `xml2eg_get`. This subroutine is polymorphic and can read strings, and scalar and arrays of integers, single and double precision floats and booleans. To read a particular value from the xml-tree, use an xpath format (e.g. `tree/branch/leaf`). Note however, that the xml2eg format assumes that the root level is called `parameters` and this root level is implicit in the xpath. Thus, while the absolute xpath would be `/parameters/tree/branch/leaf` the value of the leaf is access by requesting `tree/branch/leaf`.

The interface to `xml2eg_get` is:

```
subroutine xml2eg_get(xml2eg_document, path, out, errorflag)
type(type_xml2eg_document) :: xml2eg_document
character(len=*), intent(in) :: path
<OutType>, intent(out) :: out
logical, optional :: errorflag
```

Here `<OutType>` is either `integer`, `integer, dimension(:)`, `real(r4)`, `real(r4) dimension(:)`, `real(r8)`, `real(r8), dimension(:)`, , `boolean`, `boolean, dimension(:)`, , `character(:)`, where `r4` and `r8` are defined in the module `xmllib_types`. The most simple usage of xml2eg is then:

```
use xml2eg_mdl, only: xml2eg_parse_memory, xml2eg_get, &
     type_xml2eg_document
type(type_xml2eg_document) :: doc
character(len=132), dimension(:), pointer ::parameters
integer :: val
call xml2eg_parse_memory( parameters , doc )
call xml2eg_get(doc , 'tree/branch/leaf' , val)
```

The xml2eg interface also includes error handling. When calling `xml2eg_get` one may provide a fourth (optional) argument, which is a output argument that returns a boolean error flag (true is the reading failed and false it is was successful).

```
logical :: execution_error
...
call xml2eg_get(doc , 'tree/branch/leaf' , val, execution_error)
if (execution_error) then
     ...
end if
```

Once finished reading the xml-document all allocated data has to be freed. This is achieved by calling `xml2eg_free_doc`.

```
call xml2eg_free_doc(doc)
```

3.3 The Xpath interface

It is not recommended to do any new implementations using the Xpath interface. For examples of how to use this interface, see chapter 1.

3.4 The Konz interface

It is not recommended to do any new implementations using the Konz interface. For examples of how to use this interface, see chapter 1.

Chapter 4

Brief history of XMLLIB versions

The XMLLIB was originally develop by Christian Konz, who wrote the Konz interface within the EFDA-ITM environment, i.e. using CPOs. An extension to this interface then later written by Michal Owsiak and Denis Kalupin, who provided a routine `find_parameter`, which greatly simplified the parsing. On top of the `find_parameter` subroutine, Thomas Jonsson built the Xpath interface to even further simplify the parsing. The final component of the XMLLIB, the xml2eg interface, was provided by Edmondo Giovannozzi.

All this development was done within the EFDA-ITM environment (later renamed EUROfusion-WPCD) with a dependence on the CPOs. With the development of the ITER IMAS, a branched of the XMLLIB library was developed that provided an IDS based interface.

The most recent step (as of september 2018) in the evolution of XMLLIB was to develop a version that was completely independent of the IDS and CPO data structures. The first version of IDS and CPO independent version is based on the ITER git repository and is tagged `3.0.0`. Unfortunately, there are versions of XMLLIB, stored in the WPCD svn repository, that have version numbers `3.?` that are still dependent on the CPOs.

Chapter 5

Switching to version 3.0.0, independent of IDSs and CPOs

The 3.0.0 version of XMLLIB is independent of both the IDSs and CPOs. Earlier version of XMLLIB include all git-versions with version number below 3.0.0 (dependent on the IDSs) and all version stored in the WPCD svn repository (dependent on the CPOs). Here follows instructions for how to switch from the IDS or CPO dependent to version 3.0.0.

5.1 Real and integer precision

In IDS and CPO dependent versions of XMLLIB all subroutines use real and integer precision, as well as string lengths, defined within the IDS and CPO definitions. In order to move to a dataversion independent XMLLIB these types have been replaced by types defined within the XMLLIB module `xmllib_types`. The translations between declarations within different XMLLIB version is provided in the table below.

	IDS	CPO	XMLLIB 3.0.0
Module	ids_schemas	euitm_schemas	xmllib_types
Integer	integer(ids_int)	integer(itm_i4)	integer(int4)
Float	real(ids_real)	real(euitm_r8)	real(r8)
Strings	character(132)	character(ids_string_length)	character(132)

5.2 Reading XML information from file

As describe in 3.1, xml data can be read from file in two ways, using `fill_param`, or `file2buffer`. While the latter has always been independent of the CPOs and IDSs, the latter used to be CPO/IDS dependent. The typical usage of `fill_param` in an IDS environment is as follows:

```
use ids_schemas, only: ids_parameters_input
use xml_file_reader, only: fill_param
type(ids_parameters_input) :: param
call fill_param( param, 'input.xml' , 'input.xsd' , 'input_default.xml' )
```

The corresponding usage in a CPO environment is:

```
use euitm_schemas, only: type_param
use xml_file_reader, only: fill_param
type(type_param) :: param
```

```
call fill_param( param, 'input.xml' , 'input.xsd' , 'input_default.xml' )
```

The corresponding usage in XMLLIB 3.0.0 environment is (as described in 3.1):

```
use xml_file_reader, only: fill_param
character(132), pointer :: param_xml(:)     => NULL()
character(132), pointer :: param_xsd(:)     => NULL()
character(132), pointer :: param_default(:) => NULL()
call fill_param( param_xml , param_xsd , param_default , &
    'input.xml' , 'input.xsd' , 'input_default.xml' )
```

5.3 Parsing XML data

The xml2eg interface for parsing data is already completely independent of the IDSs and the CPOs.

The Xpath and Konz interfaces for parsing data depend on the IDSs and the CPOs in only one place, in the call to `imas_xml_parse` and `euitm_xml_parse`, respectively. Such calls using an IDS dependent version may read

```
use ids_schemas, only: type_parameters_input
use imas_xml_parser, only: tree, imas_xml_parse
type (type_param), intent(in) :: code_parameters
type(tree) :: parameter_list
call euitm_xml_parse(code_parameters, 0, parameter_list)
```

The corresponding call in a CPO dependent version is

```
use euitm_schemas, only: type_param
use euitm_xml_parser, only: tree, euitm_xml_parse
type (type_param), intent(in) :: code_parameters
type(tree) :: parameter_list
call euitm_xml_parse(code_parameters, 0, parameter_list)
```

In XMLLIB 3.0.0 the corresponding call is

```
use xmllib_parser, only: tree, xmllib_parse
character(132), pointer :: param_xml(:)
character(132), pointer :: param_xsd(:)
character(132), pointer :: param_default(:)
type(tree) :: parameter_list
call xmllib_parse(param_xml, param_xsd, param_default, 0, parameter_list)
```

Chapter 6

Examples

The following examples are taken from the directory `examples/` inside `ssh://git@git.iter.org/lib/xmllib.git`.

6.1 Example 1 for the xml2eg interface

6.1.1 The file `data.xml`

```
<parameters>
  <some_int> 13 </some_int>
  <some_real> 21.00 </some_real>
  <myfamily>
    <dad>
      <age>50</age>
      <name>Steve</name>
    </dad>
    <mum>
      <age>48</age>
      <name>Eve</name>
    </mum>
  </myfamily>
  <anotherfamily>
    <dad>
      <age>30</age>
      <name>Knut</name>
    </dad>
    <mum>
      <age>33</age>
      <name>Anna</name>
    </mum>
  </anotherfamily>
</parameters>
```

6.1.2 The file `data.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
   <xs:annotation>
     <xs:documentation>Code parameters for Nuclearsim
```

```
                    (nuclear reaction rates for thermal plasmas)</xs:documentation>
  </xs:annotation>
  <xs:element name="parameters">
    <xs:complexType>
      <xs:all>
        <xs:element name="some_int" type="xs:integer"/>
        <xs:element name="some_real" type="xs:float"/>
        <xs:element name="myfamily" type="family"/>
        <xs:element name="anotherfamily" type="family"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="family">
    <xs:all>
      <xs:element name="dad" type="person"/>
      <xs:element name="mum" type="person"/>
    </xs:all>
  </xs:complexType>
  <xs:complexType name="person">
    <xs:all>
      <xs:element name="age" type="xs:integer"/>
      <xs:element name="name" type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:schema>
```

6.1.3 The file `sample.f90`

```
program sample

  use f90_file_reader, only: file2buffer
  use xml2eg_mdl, only: xml2eg_parse_memory, xml2eg_get, &
       type_xml2eg_document, xml2eg_free_doc

  implicit none

  type(type_xml2eg_document) :: doc
  character(len=132), pointer :: buffer(:)

  character(30) :: str
  character(30) :: name
  integer :: test_int
  real(8) :: test_real
  integer :: io_unit = 1
  logical :: errorflag

  ! Here we read the xml-file and the schema from file.
  ! In ITM codes this call is typically done outside the actor, in the wrapper.
  ! The output from the call is the codeparam structure, which is the one
  ! used as input to ITM actors.
  call file2buffer('data.xml', io_unit, buffer)

  ! This call translated (parses) the xml data, stored in the string
  ! codeparam%parameters, into the DOM format used by libxml2.
```

```
! The output is the DOM document "dom".
call xml2eg_parse_memory( buffer , doc )

! Below fetch data fields from the DOM document using the xml2eg_get,
! which is overloads reading routines for strings, reals, doubles and integers,
! as well as vectors of reals, doubles and integers.
call xml2eg_get(doc , 'some_int' , test_int)
write(0,*)'First we read a single integer:',test_int

call xml2eg_get(doc , 'some_real' , test_real)
write(0,*)'...and a single real:',test_real

write(0,*)''
write(0,*)'Next lets test reading structured data, here using multiple ', &
   'instances of a "family" type:'
call xml2eg_get(doc , 'myfamily/dad/name' , name)
write(0,*)'  myfamily/dad/name=',name

call xml2eg_get(doc , 'myfamily/dad/age' , test_int)
write(0,*)'  myfamily/dad/age=',test_int

call xml2eg_get(doc , 'myfamily/mum/name' , name)
write(0,*)'  myfamily/mum/name=',name

call xml2eg_get(doc , 'myfamily/mum/age' , test_int)
write(0,*)'  myfamily/mum/age=',test_int

call xml2eg_get(doc , 'anotherfamily/dad/name' , name)
write(0,*)'  anotherfamily/dad/name=',name

call xml2eg_get(doc , 'anotherfamily/dad/age' , test_int)
write(0,*)'  anotherfamily/dad/age=',test_int

call xml2eg_get(doc , 'anotherfamily/mum/name' , name)
write(0,*)'  anotherfamily/mum/name=',name

call xml2eg_get(doc , 'anotherfamily/mum/age' , test_int)
write(0,*)'  anotherfamily/mum/age=',test_int

write(0,*)
str='does/not/exists'
write(0,*)'When attempting reading a path that does not exists in the ', &
     'xml (e.g. "',trim(str),'") then...'
write(0,*)'    (a) The following error message is generated:'

call xml2eg_get(doc , str , test_int, errorflag)

write(0,*)'    (b) for integers you recieve the answer: ',test_int
write(0,*)'    (c) you may recieve an optional error-flag (logical) with value: ', &
     errorflag

call xml2eg_free_doc(doc)
deallocate(buffer)
```

```
end program sample
```

6.2 Example 2 for the xml2eg interface

6.2.1 The file actor_example.xml

```
<parameter>
  <branch_1>
    <my_integer> 153</my_integer>
  </branch_1>
  <branch_2>
    <my_real1>3.14</my_real1>
    <my_real>1.6022e-19</my_real>
  </branch_2>
</parameter>
```

6.2.2 The file actor_example.f90

```
subroutine actor_example(equilibrium, core_profiles, codeparam)

  use ids_schemas, only: ids_real, ids_int, ids_equilibrium, &
     ids_core_profiles, ids_parameters_input
  use iso_c_binding, only: c_double, c_int

  implicit none

  type(ids_equilibrium), intent(in) :: equilibrium
  type(ids_core_profiles), intent(out) :: core_profiles
  type(ids_parameters_input), intent(in) :: codeparam

  ! Internal
  integer(ids_int) :: value_int       ! IDS compatible variable
  real(ids_real)   :: value_real1     ! IDS compatible variable
  real(ids_real)   :: value_real2     ! IDS compatible variable
  integer(c_int)   :: tmp_value_int   ! XML2EG compatible variable
  real(c_double)   :: tmp_value_real1 ! XML2EG compatible variable
  real(c_double)   :: tmp_value_real2 ! XML2EG compatible variable

  interface
     subroutine assign_codeparam(codeparam_string, value_int, value_real1, value_real2)
       use iso_c_binding, only: c_double, c_int
       character(len=132), pointer :: codeparam_string(:)
       integer(c_int) :: value_int
       real(c_double) :: value_real1
       real(c_double) :: value_real2
     end subroutine assign_codeparam
  end interface

  call assign_codeparam(codeparam%parameters_value, &
       tmp_value_int, tmp_value_real1, tmp_value_real2)
  value_int  = int(  tmp_value_int  , ids_int  )
  value_real1 = real( tmp_value_real1 , ids_real )
  value_real2 = real( tmp_value_real2 , ids_real )
```

```
  write(*,*)'Hello!'
  write(*,*)'Integer value, 153_ids_int = ', value_int
  write(*,*)'Real value, 3.14_ids_real          = ', value_real1
  write(*,*)'Real value, 1.6022000E-19_ids_real = ', value_real2

end subroutine actor_example

!--------------------------------------------------------------------------------

subroutine assign_codeparam(codeparam_string, value_int, value_real1, value_real2)

  use xml2eg_mdl, only: xml2eg_parse_memory, xml2eg_get, type_xml2eg_document, &
      xml2eg_free_doc, set_verbose
  use iso_c_binding, only: c_double, c_int

  implicit none

  ! Input/Output
  character(len=132), pointer :: codeparam_string(:)
  integer(c_int) :: value_int
  real(c_double) :: value_real1
  real(c_double) :: value_real2

  ! Internal
  type(type_xml2eg_document) :: doc

  ! Parse the "codeparam_string". This means that the data is put into a document "doc"
  call xml2eg_parse_memory( codeparam_string  , doc )
  call set_verbose(.TRUE.) ! Only needed if you want to see what's going on in the parsing

  ! Extract data in "doc" at position "branch_1/my_integer" and store it in "value_int"
  call xml2eg_get( doc , 'branch_1/my_integer' , value_int  )

  ! Extract data in "doc" at position "branch_1/my_real1" and store it in "value_real1"
  call xml2eg_get( doc , 'branch_2/my_real1'    , value_real1 )

  ! Extract data in "doc" at position "branch_1/my_real" and store it in "value_real2"
  call xml2eg_get( doc , 'branch_2/my_real'    , value_real2 )

  ! Make sure to clean up after you!!
  ! When calling "xml2eg_parse_memory" memory was allocated in the "doc" object.
  ! This memory is freed by "xml2eg_free_doc(doc)"
  call xml2eg_free_doc(doc)

end subroutine assign_codeparam
```

6.2.3 The file prog_actor_example.f90

```
program prog_actor_example

  use ids_schemas, only: ids_equilibrium, ids_core_profiles, ids_parameters_input
  use f90_file_reader, only: file2buffer
```

```
  type(ids_equilibrium) :: equilibrium
  type(ids_core_profiles) :: core_profiles
  type(ids_parameters_input) :: codeparam
  integer :: iounit = 1
  character(len=132), pointer :: buffer(:)

  interface
     subroutine actor_example(equilibrium, core_profiles, codeparam)
       use ids_schemas, only: ids_equilibrium, ids_core_profiles, ids_parameters_input
       type(ids_equilibrium), intent(in) :: equilibrium
       type(ids_core_profiles), intent(out) :: core_profiles
       type(ids_parameters_input), intent(in) :: codeparam
     end subroutine actor_example
  end interface

  write(*,*)'Reading actor_example.xml...'
  call file2buffer('actor_example.xml',iounit, codeparam%parameters_value)
  write(*,*)'call actor_example...'
  call actor_example(equilibrium, core_profiles, codeparam)

end program prog_actor_example
```