Python API for Fleur XML files: Design

Henning Janßen

Nov 17, 2022

CONTENTS

This site aims to explain the design of the current functionality for working with `inp.xml` and `out.xml` files in both `masci-tools` and `aiida-fleur`

Ultimately this document explains the background behind the following code blocks for loading information from `.xml` files

AiiDA

```python
from aiida import plugins

FleurinpData = plugins.DataFactory('fleur.fleurinp')
fleurinp = FleurinpData('inp.xml')

structure = fleurinp.get_structuredata()
lapw_parameters = fleurinp.get_parameterdata()
```

non-AiiDA

```python
from masci_tools.io.io_fleurxml import load_inpxml
from masci_tools.util.xml.xml_getters import get_structure_data, get_parameter_data

xmltree, schema_dict = load_inpxml('inp.xml')

structure_information = get_structure_data(xmltree, schema_dict)
lapw_parameters = get_parameter_data(xmltree, schema_dict)
```

and modifying `.xml` files

AiiDA

```python
from aiida_fleur.data.fleurinpmodifier import FleurinpModifier

modifier = FleurinpModifier(fleurinp)
modifier.set_inpchanges({'kmax': 5.0, 'l_soc': True})
modifier.set_species('all-Fe', {
                        'cutoffs': {'lmax': 18}
                     },
                     filters={'species':{
                        'mtSphere': {'radius': {'>': 2.0}}
                     }})

fleurinp = modifier.freeze()
```

non-AiiDA

```python
from masci_tools.io.fleurxmlmodifier import FleurXMLModifier

modifier = FleurXMLModifier()
modifier.set_inpchanges({'kmax': 5.0, 'l_soc': True})
modifier.set_species('all-Fe', {
                        'cutoffs': {'lmax': 18}
                     },
                     filters={'species':{
                        'mtSphere': {'radius': {'>': 2.0}}
                     }})

xmltree, _ = modifier.modify_xmlfile('inp.xml')
```

Note: This document does not try to explain the AiiDA framework itself but only the parts relevant for the XML functionality. For more information on AiiDA refer to it's documentation

CHAPTER

ONE

PROBLEM

The main Fleur input/output files are in a XML format since version `v27`. This provides external tools with the ability to retrieve information from these or modify the input file in a well defined manner.

However, changes in the structure of these XML formats produces a non-trivial amount of maintenance that needs to happen in external packages. Especially the AiiDA plugin for the Fleur code was required to publish a release with a lot of small compatibility changes for each release of the code, limiting the ability to support more Fleur releases with the same version of the plugin. For example in releases up to version `1.1.4` a function was implemented hardcoding the needed information about the `inp.xml` file in this function and the parser for the `out.xml` file also hardcoded a lot of the file structure (see here)

The XML functionality in `masci-tools` aims to provide a way to centralize this effort and make the extension and implementation of new features easier.

CHAPTER

TWO

DESIGN PRINCIPLES

- The previous implementation of XML file parsers in `aiida-fleur` made use of XML `XPath` to retrieve information from XML files. For performance reasons these are always absolute paths completely definining the location of the desired information. The use of absolute XPaths should also be preferred in the new implementation
- Users should not be required to know the complete `XPath` to be able to retrieve information, but ideally and if possible only the name of the last node in the path. The complete Path would be special knowledge and new functionality should be much closer to the way `set_inpchanges` works
- Maintenance for including changes in the XML file structure with new Fleur releases should be minimal
- All AiiDA functionality directly manipulating/reading XML files should have an equivalent way of doing the same thing without AiiDA. `aiida-fleur` should only provide a light wrapper adding AiiDA specific functionality around the functions in `masci-tools` for these cases. This not only reduces the barrier of entry for people experimenting with it since no large AiiDA environment has to be set up. It also enables the reusage of this code for adding Fleur IO capability to other packages

CHAPTER

THREE

FLEUR XML FILE STRUCTURE

The structure of the Fleur XML files is defined in so called XML schema files. There are two files

- `FleurInputSchema.xsd` defines the complete structure of the `inp.xml`
- `FleurOutputSchema.xsd` defines the complete structure of the `out.xml`

Note: Since the `out.xml` contains a copy of the used input to make it useful as a standalone file the `FleurOutputSchema.xsd` also explicitly includes the `FleurInputSchema.xsd`

The structure of the XML files is very specific.

- A large amount of the data is stored in XML attributes
- The `out.xml` file makes more use of the text for XML tags.
- The XML files are non-recursive (with the exception of the `timing` section in the `out.xml`)
- The data types in the XML are limited to:
  - `switches`: Booleans expressed as either `T` or `F` strings
  - `strings`
  - `integers`
  - `floats`
  - `mathematical expressions`: Simple strings that Fleur can evaluate
  - `complex numbers`

CHAPTER

FOUR

LOADING XML FILES

The first step in doing anything with the Fleur XML files is parsing them into a datastructure in the python runtime. For this we use the `lxml` library. This library provides the very popular `ElementTree` API originally from the `python stdlib` and has complete support for `XPath1.0` and `XInclude`. The latter is used to split up input files into more digestable chunks for users. In addition it uses the `libxml2` library under the hood, which is the same library used by the Fleur code itself.

Using lxml for untrusted input

For XML files from untrusted sources special care must be taken to avoid security problems. A general guide for using `lxml` in these cirumstances can be found here

Some features used by the Fleur XML files have to be limited in these cases:

- `XInclude`: Can in principle retrieve files from other sources on the web

At the moment there are only a couple measures taken to make the API in `masci-tools` safer but it is definitely not safe for use directly as a web service

Example of parsing XML files with `lxml`

Listing 1: test.xml

```xml
<root>
    <child>Child 1</child>
    <child>Child 2</child>
    <another>Child 3</another>
</root>
```

```python
from lxml import etree

xmltree = etree.parse('test.xml')

#The xmltree now contains the complete information from the XML file
root = xmltree.getroot()
print(f'Root Tag: {root.tag}')

text = [node.text for node in root.findall('child')]
print(f"Text of 'child' elements: {text}")

text = root.xpath('//child/text()')
print(f"Text of 'child' elements: {text}")
```

```
Root Tag: root
Text of 'child' elements: ['Child 1', 'Child 2']
Text of 'child' elements: ['Child 1', 'Child 2']
```

The code cell above uses two equivalent methods of searching the XML tree. First we use a method of the `ElementTree` API called `findall` which searches for tags with the given name. In the second way we use a the called XPath standard for specifying a path through the XML tree simlar to a file system path.

The latter is used in the majority of cases in the following implementations. It allows us to be very specific with what we want to find and it allows for more complex syntax to *filter the results*. It also allows us to express a unique location in the XML file with a single string, enabling us to define mappings between the locations in the XML file and their attached properties. Further as long as a XML file is not recursive, i.e. a tag cannot contain itselve, the number of the unique locations defined by absolute XPaths without complex syntax is finite.

CHAPTER

FIVE

USING THE XML SCHEMA FILE

The XML Schema files provide a great way of programmatically discovering the structure of the `.xml` files and the types of the used attributes/text. The Schema files can be used in two ways by the `lxml` library. First an `etree.XMLSchema` object can be constructed to validate given XML files against this schema.

```python
from lxml import etree

schema = etree.XMLSchema(file='schemas/FleurInputSchema.xsd')

xmltree = etree.parse('example_files/inp_valid.xml')
print(f'Is this file valid? {schema(xmltree)}')
```

```
Is this file valid? True
```

```python
from lxml import etree

schema = etree.XMLSchema(file='schemas/FleurInputSchema.xsd')

xmltree = etree.parse('example_files/inp_invalid.xml')
print(f'Is this file valid? {schema(xmltree)}')
```

```
Is this file valid? False
```

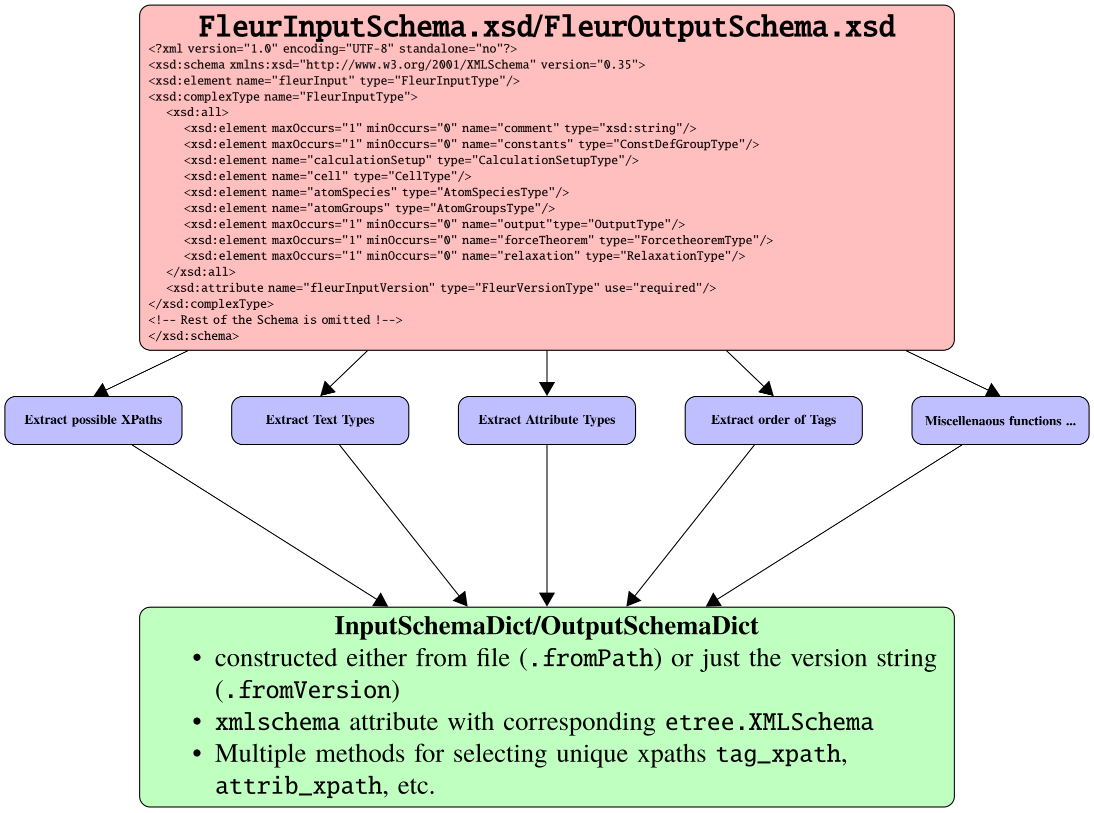
Or the file can be used like a normal XML file. The following example retrieves how many different complex types are defined in the schema.

Note: In Schema files all tags are prefixed with a special namespace `xsd`

```python
from lxml import etree

xmltree = etree.parse('schemas/FleurInputSchema.xsd')

namespaces = {'xsd': 'http://www.w3.org/2001/XMLSchema'}
n_types = xmltree.xpath('count(/xsd:schema/xsd:complexType)', namespaces=namespaces)

print(f'The input schema has {int(n_types)} complex types')
```

```
The input schema has 96 complex types
```

We employ the second way of handling XML Schemas to discover their structure. The found information is stored in dictionary like structures called `InputSchemaDict` and `OutputSchemaDict`


FleurInputSchema.xsd/FleurOutputSchema.xsd
Extract possible XPaths
Extract Text Types
Extract Attribute Types
Extract order of Tags
Miscellenaous functions ...
InputSchemaDict/OutputSchemaDict

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" version="0.35">
<xsd:element name="fleurInput" type="FleurInputType"/>
<xsd:complexType name="FleurInputType">
  <xsd:all>
    <xsd:element maxOccurs="1" minOccurs="0" name="comment" type="xsd:string"/>
    <xsd:element maxOccurs="1" minOccurs="0" name="constants" type="ConstDefGroupType"/>
    <xsd:element name="calculationSetup" type="CalculationSetupType"/>
    <xsd:element name="cell" type="CellType"/>
    <xsd:element name="atomSpecies" type="AtomSpeciesType"/>
    <xsd:element name="atomGroups" type="AtomGroupsType"/>
    <xsd:element maxOccurs="1" minOccurs="0" name="output"type="OutputType"/>
    <xsd:element maxOccurs="1" minOccurs="0" name="forceTheorem" type="ForcetheoremType"/>
    <xsd:element maxOccurs="1" minOccurs="0" name="relaxation" type="RelaxationType"/>
  </xsd:all>
  <xsd:attribute name="fleurInputVersion" type="FleurVersionType" use="required"/>
</xsd:complexType>
<!-- Rest of the Schema is omitted !-->
</xsd:schema>
```

- constructed either from file (`.fromPath`) or just the version string (`.fromVersion`)
- `xmlschema` attribute with corresponding `etree.XMLSchema`
- Multiple methods for selecting unique xpaths `tag_xpath`, `attrib_xpath`, etc.

Fig. 1: General structure of extracting information from XML Schema files. Blue nodes each refer to small functions working on the XML tree of the schema directly

Performance implications

Looking at this structure one problem you might spot is that we dramatically increased the number of operations needed, just to get the path to a given tag. Previously the path would just be hardcoded meaning no performance impact at all. With the new implementation a XML file maybe larger than the actual input file has to be parsed before gaining access to e.g. the path to a tag.

To manage the impact of this there are multiple layers of reusing results, i.e. caching in the construction of the `SchemaDict` objects.

1. The individual functions parsing the schema file use a lot of similar `XPath` expressions over and over again (on the order of ~1000 queries). The results of these queries are cached
2. If a `SchemaDict` is constructed using the `fromVersion` method the SchemaDict is only constructed on the first run. On subsequent calls with the same version string the cached object is returned

The `load_inpxml` and `load_outxml` functions in `masci_tools.io.io_fleurxml` provide this schema dictionary together with the parsed xml file just by giving the filepath to a given XML file. The `aiida-fleur` `FleurinpData` class has a wrapper method for loading and validating `inp.xml` files, which is called, when any instance is instanitated with a file named `inp.xml`.

All functionality in the following sections is built on top of these objects to remove the need for hardcoding the structure of the Fleur XML files in multiple places

5.1 Folder structure

Below we show the folder structure of the subpackage in `masci-tools` implementing the parsing of the XML Schema described above

```
fleur_schema
├── 0.27
│   └── FleurInputSchema.xsd
├── 0.28
│   └── FleurInputSchema.xsd
├── 0.29
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── 0.30
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── 0.31
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── 0.32
│   └── FleurInputSchema.xsd
├── 0.33
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── 0.34
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── 0.35
│   ├── FleurInputSchema.xsd
│   └── FleurOutputSchema.xsd
├── __init__.py
├── fleur_schema_parser_functions.py
├── inpschema_todict.py
├── outschema_todict.py
└── schema_dict.py
```

There are two parts to this implementation. First we have the python files that contain the functionality and the subfolders which contain the data in form of the Fleur XML schema files for each file version.

- `fleur_schema_parser_functions.py` Contains all the blue node functions from above
- `inpschema_todict.py` Collects and executes the parsing functions for the inp.xml
- `outschema_todict.py` Collects and executes the parsing functions for the out.xml
- `schema_dict.py` Defines the InputSchemaDict and OutputSchemaDict classes. Calls the functions in `inpschema_todict.py` and `outschema_todict.py` to construct the instances
- `__init__.py` Exports the important classes so that they are available under `masci_tools.io.parsers.fleur_schema` directly

The file schemas are put into the subfolders with their corresponding file version number (appears in the schema under `/xsd:schema/@version`). The names of these subfolders are limited to be numbers separated by dots and correspond exactly to the argument that has t be given to the `fromVersion` method to get `SchemaDict` instances for these versions.

5.2 Selecting XPaths

Below is a simple example of getting the complete Xpath for a given tag name

```
from masci_tools.io.parsers.fleur_schema import InputSchemaDict

schema_dict = InputSchemaDict.fromVersion('0.35')
print(schema_dict.tag_xpath('xcfunctional'))
```

```
/fleurInput/calculationSetup/xcFunctional
```

```
from masci_tools.io.parsers.fleur_schema import InputSchemaDict

schema_dict = InputSchemaDict.fromVersion('0.31')
print(schema_dict.tag_xpath('xcfunctional'))
```

```
/fleurInput/xcFunctional
```

Note: The tag name given in the method is case-insensitive `XCFUNCTIONAL` would also work

5.3 Working with types

Accessing the collected types of attribute/text

```
from masci_tools.io.parsers.fleur_schema import InputSchemaDict

schema_dict = InputSchemaDict.fromVersion('0.35')
print(schema_dict['attrib_types']['alpha'])
print(schema_dict['text_types']['kpoint'])
```

```
[AttributeType(base_type='float', length=1), AttributeType(base_type='float_expression',␣
↪length=1)]
[AttributeType(base_type='float_expression', length=3)]
```

`masci_tools.util.xml.converters` provides functions for robustly using these types

```
from masci_tools.io.parsers.fleur_schema import InputSchemaDict
from masci_tools.util.xml.converters import convert_from_xml, convert_to_xml

schema_dict = InputSchemaDict.fromVersion('0.35')
print(convert_from_xml('16.500000000', schema_dict, 'alpha'))
print(convert_from_xml('16.500000000*Pi', schema_dict, 'alpha')) #mathematical␣
↪expressions

print(convert_to_xml([0.0,1.0,2.0], schema_dict, 'kpoint', text=True))
```

```
(16.5, True)
(51.83627878423159, True)
(' 0.0000000000000  1.0000000000000  2.0000000000000', True)
```

Note: The boolean second return value indicates whether the conversion was successful

CHAPTER SIX

RETRIEVING INFORMATION FROM XML FILES

There are two levels of functions building on top of the schema functionality for retrieving information from XML files

- Functions, that retrieve and convert values for one given tag/attribute from the xml file
  - `evaluate_attribute`, `evaluate_text`, `evaluate_tag`, `evaluate_parent_tag`, `evaluate_single_value`, `tag_exists`, `attrib_exists`, `get_number_of_nodes`
- `FleurXMLContext` is a convenience class to bundle the above functions
- Functions, that return a collection of values for a clearly defined property of the calculation
  - `get_fleur_modes`: Dictionary with general switches and numbers identifying the *kind* of Fleur calculation
  - `get_cell`: Return the braivais matrix and periodicity
  - `get_structure_data`: Return the atom position, symbols, braivais matrix and periodicity
  - `get_parameter_data`: Return dict with additional LAPW parameters (cutoffs, ...)
  - `get_kpoints_data`: Get Kpoint information in arrays
  - `get_nkpts`: Get the numnber of kpoints used in the calculation (mainly for optimizing parallelization)
  - `get_special_kpoints`: Get labelled kpoints in a kpoint set
  - `get_symmetry_information`: Get the symmetry operations in the calculation

The second set of functions uses the functions above to avoid hardcoding `XPath`

6.1 Usage of universal functions

First let's see a few examples of usage for the universal functions that can operate on any part of the XML file, like `evaluate_attribute`. The basic usage only requires the name of the attribute to obtain in this case. This pattern holds for any of the functions in `masci_tools.util.schema_dict_util`

```
from masci_tools.io.fleur_xml import load_inpxml
from masci_tools.util.schema_dict_util import evaluate_attribute

xmltree, schema_dict = load_inpxml('example_files/inp_valid.xml')

print(f"Number of spins: {evaluate_attribute(xmltree, schema_dict, 'jspins')}")
```

```
Number of spins: 1
```

If the attribute name allows more than one possibility an error is raised and the desired attribute can be selected using phrases that should (`contains`) or should not (`not_contains`) be included in the path in the XML file.

```
evaluate_attribute(xmltree, schema_dict, 'radius')
```

```
NoUniquePathFound: The attrib radius has multiple possible paths with the current␣
↪specification.
contains: None, not_contains: None, exclude []
These are possible: ['/fleurInput/atomGroups/atomGroup/mtSphere/@radius', '/fleurInput/
↪atomSpecies/species/mtSphere/@radius']
```

```
evaluate_attribute(xmltree, schema_dict, 'radius', contains='species')
```

```
2.17
```

```
#The result is empty since the example file has no mtRadius attribtues specified
#on the atom groups
evaluate_attribute(xmltree, schema_dict, 'radius', not_contains='species')
```

```
ValueError: No values found for attribute radius
```

6.2 Relative XPaths

Another approach to ditinguish the two paths possible in the example above is to change the `xmltree` argument out with an element from the XML tree, from which the choice is no longer ambiguous.

```
from masci_tools.util.schema_dict_util import eval_simple_xpath
species = eval_simple_xpath(xmltree, schema_dict, 'atomSpecies')
```

Now when looking forward from the `atomSpecies` node, there is only one `radius` attribute so the `evaluate_attribute` function works without further specifications.

```
evaluate_attribute(species, schema_dict, 'radius')
```

```
2.17
```

6.3 FleurXMLContext

The FleurXMLContext class allows to bundle multiple evaluations and make them more concise. It holds aliases to all the universal functions, with the `xmltree` and `schema_dict` arguments already filled in.

- `attribute`: `evaluate_attribute`
- `text`: `evaluate_text`
- `all_attributes()`: `evaluate_tag`
- `parent_attributes()`: `evaluate_parent_tag`
- `single_value()`: `evaluate_single_value`

- `tag_exists()`: `tag_exists`
- `number_nodes()`: `get_number_of_nodes`
- `attribute_exists()`: `attrib_exists`
- `simple_xpath()`: `eval_simple_xpath`

In addition it has three methods for easily change the node from which to evaluate expressions from. These are either used in a context manager (`with` block), or in the case of `iter` in a `for` loop

- `find()`: Find the first occurrence of the tag and change to this node during the `with` block
- `iter()`: Find all occurrences of the tag and iterate through the nodes in a loop
- `nested()`: Pass in a node and inherit all the other context within the `with` block

```python
from masci_tools.io.fleur_xml import FleurXMLContext

xmltree, schema_dict = load_inpxml('example_files/inp_valid.xml')

with FleurXMLContext(xmltree, schema_dict) as root:
   print(f"Number of spins: {root.attribute('jspins')}")
   print(f"Muffin-tin radius: {root.attribute('radius', contains='species')}")

   print('Atomic positions:')
   for pos in root.iter('relpos'):
      print(f"{pos.attribute('label')}: {pos.text('relpos')}")
```

```
Number of spins: 1
Muffin-tin radius: 2.17
Atomic positions:
                   1: [0.125, 0.125, 0.125]
                   2: [-0.125, -0.125, -0.125]
```

6.3.1 Specialized functions

All the other more specialized functions only need the `xmltree` and `schema_dict` arguments to work and are located in `masci_tools.util.xml.xml_getters`. Since they are way more specific they have unique outputs. For details on this, please look into the `masci-tools` documentation

```python
from masci_tools.io.fleur_xml import load_inpxml
from masci_tools.util.xml.xml_getters import get_structure_data

xmltree, schema_dict = load_inpxml('example_files/inp_valid.xml')

atoms, cell, pbc = get_structure_data(xmltree, schema_dict)

print('Bravais matrix:')
print(cell)
print(f'Periodic boundary conditions (x,y,z): {pbc}')
print('Atomic information:')
for atom in atoms:
   print(atom)
```

```
Bravais matrix:
[[0.         2.71500111 2.71500111]
 [2.71500111 0.         2.71500111]
 [2.71500111 2.71500111 0.        ]]
Periodic boundary conditions (x,y,z): (True, True, True)
Atomic information:
AtomSiteProperties(position=[0.6787502783660522, 0.6787502783660522, 0.6787502783660522],
↪ symbol='Si', kind='Si-1')
AtomSiteProperties(position=[-0.6787502783660522, -0.6787502783660522, -0.
↪6787502783660522], symbol='Si', kind='Si-1')
```

6.4 XML getter Versioning

For different versions of the input/output the logic of the specialized XML getter functions might need to change. If the required changes are limited in size this can be achieved using the `inp_version`/`out_version` attributes of the `SchemaDict` objects.

If more involved changes are needed a mechanism is available for creating multiple variants of a function which are called based on the file version. This approach is based on decorators.

```python
from masci_tools.io.parsers.fleur_schema import schema_dict_version_dispatch

@schema_dict_version_dispatch(output_schema=False)
def example_xml_function(xmltree, schema_dict):
   """
   This is the default version of the XML getter function

   output_schema=False means we distinguish the variants by the input version
   """
   ...

@example_xml_function.register(max_version='0.31')
def alternative_xml_function(xmltree, schema_dict):
   """
   This is an alternative version of the XML getter function
   It will be called for any file with input version 0.31 or older

   In all other cases the default function is called
   """
   ...
```

CHAPTER

SEVEN

MODIFYING XML FILES

When modifying the XML files more care has to be taken. The changes made could leave the XML file in a state inconsistent with the XML schema if the modification is done in multiple steps. In the previous implementation this was solved in `aiida-fleur` by bundling multiple changes together in the `FleurinpModifier` class, which are then executed together and changes are only validated at the end.

This approach is also taken in the new implementation making use of the `SchemaDict` functionalities. However, we introduce a much more strict separation of concerns of the underlying functions modifying the XML tree to make extension and reasoning about modifications easier. The three main categories of functions are shown below. These

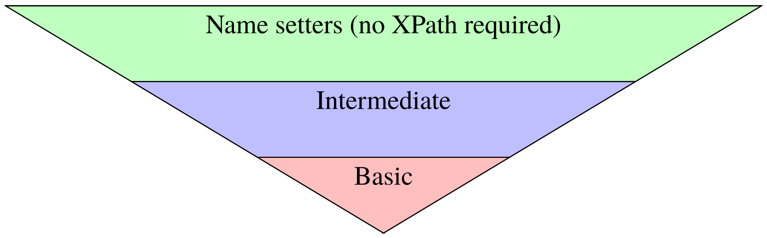


Fig. 1: Hierachy of XML setters. The width indicates that most if not all functionality should be introduced in the higher levels and reuse code in the levels below extensively

categories are mainly identified by their function signature of their first arguments:

- `Basic` setter functions: These take the XML tree and an explicit `XPath` expression and no `SchemaDict`
- `Intermediate` setter functions: These take the XML tree, the corresponding `SchemaDict` and two `XPath` expressions. One is a `XPath` that is "simple", i.e. it is absolute and has no predicates (conditions in `[]`). The second `XPath` is unrestricted but has to produce a subset of the results the "simple" `XPath` produces
- `Name` setter functions: These take the XML tree, the corresponding `SchemaDict` and require no `XPath` expression. They can however take a `XPath` expression as an optional argument

When implementing XML setters it is allowed to call different functions on the same level of modification function or a level below (according to the pyramid). It is never allowed to go back up in the levels. The reason for this is that each higher level provides more functionality that cannot be reproduced consistently when going back up. For example the `Intermediate` functions are allowed to create nested tags if they are not present in the XML file. This is possible since these functions have access to the `SchemaDict` to make sure that the order of tags is always correct and no invalid tags are created. The `Basic` functions have no guarantees for this since they do not have access to the `SchemaDict`

CHAPTER

EIGHT

FILTERING RESULTS

Almost all of the universal modifying and getting functions have an argument `filters`. This enables more involved selection of results by other values/properties of the XML file.

In order to use this feature however, one needs to know a bit more about the XPath of the property of interest. Let's say we again want to evaluate the `radius` attribute on the `species` tag. This coressponds to the following basic XPath

```
/fleurInput/atomSpecies/species/mtSphere/@radius
```

And now we can use each component of the path to add more conditions to it. A complete documentation of the possible syntax is available in the masci-tools documentation.

Here we show a few examples of the `filters` argument and their resulting XPath

Only get muffin-tin radius of iron atoms

```
filters={
   'species': {'element': 'Fe'}
}
```

```
/fleurInput/atomSpecies/species[@element='Fe']/mtSphere/@radius
```

Only get muffin-tin of atoms with lmax bigger than 8

```
filters={
   'mtSphere': {'lmax': {'>': 8}}
}
```

```
/fleurInput/atomSpecies/species/mtSphere[@lmax>8]/@radius
```

Only get muffin-tin of atoms with lmax bigger than 8 and present los

```
filters={
   'mtSphere': {'lmax': {'>': 8}},
   'species': {'has': 'lo'}
}
```

```
/fleurInput/atomSpecies/species[lo]/mtSphere[@lmax>8]/@radius
```

Multiple conditions

```
filters={
   'mtSphere': {'and': [{'lmax': {'>': 8}}, {'radius': {'<': 2}}]}
}
```

```
/fleurInput/atomSpecies/species/mtSphere[@lmax>8 and @radius<2]/@radius
```

Test for membership

```
filters={
   'species': {'element': {'in': ['Fe', 'Si']}}
}
```

```
/fleurInput/atomSpecies/species[@element='Fe' or @element='Si']/mtSphere/@radius
```

CHAPTER NINE

PREDEFINED COMPLETE FILE PARSERS

9.1 inp.xml

The parser for the `inp.xml` is very simple, since the `inp.xml` is limited in size the parser will go through all tags and attributes in the `inp.xml` file and convert them to python types according to their type. This complete representation is used in `aiida-fleur` for example to have a complete representation of the values in the database and making it queriable.

```
from masci_tools.io.parsers.fleur import inpxml_parser
from pprint import pprint

inp_dict = inpxml_parser('example_files/inp_valid.xml')

pprint(inp_dict['calculationSetup'])
```

```
{'coreElectrons': {'coretail_lmax': 0,
                   'ctail': True,
                   'frcor': False,
                   'kcrel': 0},
 'cutoffs': {'Gmax': 11.1, 'GmaxXC': 9.2, 'Kmax': 3.5, 'numbands': 0},
 'expertModes': {'gw': 0, 'isec1': 99, 'secvar': False},
 'ldaU': {'l_linMix': False, 'mixParam': 0.05, 'spinf': 1.0},
 'magnetism': {'jspins': 1, 'l_noco': False, 'lflip': False, 'swsp': False},
 'scfLoop': {'alpha': 0.05,
             'imix': 'Anderson',
             'itmax': 6,
             'maxIterBroyd': 99,
             'minDistance': 0.0,
             'spinf': 2.0},
 'soc': {'l_soc': False, 'phi': 0.0, 'spav': False, 'theta': 0.0},
 'xcFunctional': {'name': 'pbe', 'relativisticCorrections': False}}
```

9.2 out.xml

The parser for the output XML file needs to be a lot more selective. Since the size of the calculation can be massive, it not only means that the correcponding output dictionary would also be massive and the important information would be lost. In most cases the user is only interested in selected properties from the last converged SCF iteration. However, the properties of interest will depend on the kind of calculation that is performed. Therefore the design needs to be a lot more modular.

The basic idea is to break down the parsing of the file into different blocks that can be treated as one entity. Examples for this are

- Determine total energy and individual contributions to the total energy
- Determine the fermi energy
- Determine magnetic or orbital moments
- Determine forces
- . . .

Note: These tasks are defined in the form of a dictionary in `masci_tools.io.parser.fleur.default_parse_tasks`. For a complete reference on the possibilities look into the masci-tools documentation. Below an example for retrieveing the total energy

```
total_energy_definition = {
   'energy_hartree': {
      'parse_type': 'singleValue',
      'path_spec': {
            'name': 'totalEnergy'
      }
   },
}
```

In the beginning now the `get_fleur_modes` function (see above) is used to determine the kind of calculation. From this the tasks to perform can be deduced. The structure of the `out.xml` parser is sketched. Below an example of the kind of information parsed is shown

```
from masci_tools.io.parsers.fleur import outxml_parser
from pprint import pprint

#By default the last iteration is parsed
#This can be modified with the iteration_to_parse argument
#e.g. iteration_to_parse='all' will parse all SCF iterations
result = outxml_parser('example_files/out.xml')
pprint(result)
```

```
{'bandgap': 0.1480235781,
 'bandgap_units': 'eV',
 'charge_den_xc_den_integral': -111.7983059673,
 'creator_name': 'fleur 32',
 'creator_target_architecture': 'GEN',
 'density_convergence': [0.0056284999, 0.003709981],
 'density_convergence_units': 'me/bohr^3',
```

(continues on next page)

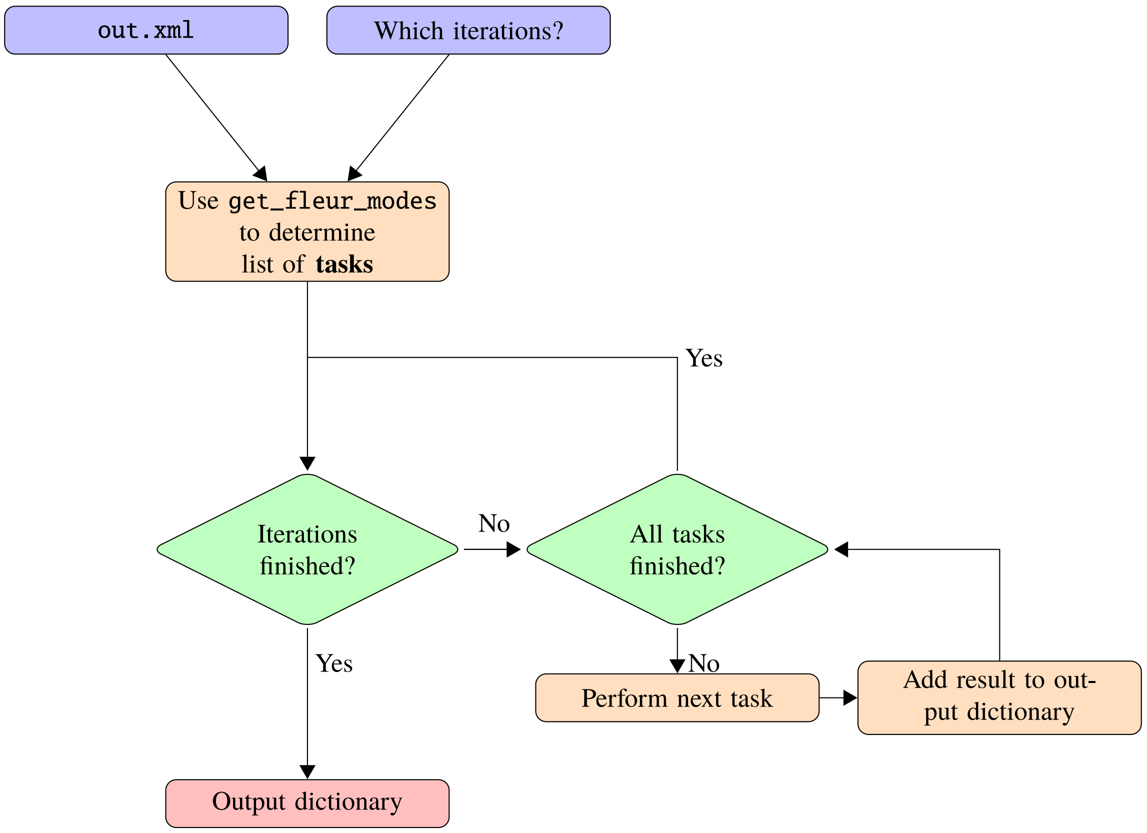


Fig. 1: Structure of the out.xml parser

(continued from previous page)

```
'end_date': {'date': '2020/12/10', 'time': '16:51:58'},
'energy': -69269.485168433,
'energy_core_electrons': -1476.8200550748,
'energy_hartree': -2545.6066273965,
'energy_hartree_units': 'Htr',
'energy_units': 'eV',
'energy_valence_electrons': 3.7137334342,
'fermi_energy': 0.3263945678,
'fermi_energy_units': 'Htr',
'film': False,
'fleur_modes': {'band': False,
                'bz_integration': 'hist',
                'cf_coeff': False,
                'dos': False,
                'film': False,
                'force_theorem': False,
                'greensf': False,
                'jspin': 2,
                'ldahia': False,
                'ldau': False,
                'noco': True,
                'plot': False,
                'relax': False,
                'soc': True,
                'spex': 0},
'gmax': 10.2,
'input_file_version': '0.34',
'kmax': 3.4,
'magnetic_moments': [1.7481314169, 1.7484083179],
'magnetic_moments_spin_down_charge': [2.6541041481, 2.6540102684],
'magnetic_moments_spin_up_charge': [4.4022355649, 4.4024185864],
'number_of_atom_types': 2,
'number_of_atoms': 2,
'number_of_iterations': 20,
'number_of_iterations_total': 20,
'number_of_kpoints': 3,
'number_of_species': 2,
'number_of_spin_components': 2,
'number_of_symmetries': 16,
'orbital_magnetic_moments': [0.0541958344, 0.0541948912],
'orbital_magnetic_moments_spin_down_charge': [0.0839236153, 0.083921618],
'orbital_magnetic_moments_spin_up_charge': [-0.0297277809, -0.0297267268],
'output_file_version': '0.34',
'overall_density_convergence': 0.0060794372,
'spin_density_convergence': 0.0073435947,
'spin_dependent_charge_interstitial': [0.9585356, 0.9585064],
'spin_dependent_charge_mt_spheres': [26.7903595, 23.2925985],
'spin_dependent_charge_total': [27.7488951, 24.2511049],
'start_date': {'date': '2020/12/10', 'time': '16:51:52'},
'sum_of_eigenvalues': -1473.1063216407,
'title': 'Fe fcc 2',
'total_charge': 52.0000000742,
```

(continues on next page)

(continued from previous page)

```
 'total_magnetic_moment_cell': 3.497790199999997,
 'ts_energy': 0.006292567762057734,
 'ts_energy_units': 'eV',
 'walltime': 6,
 'walltime_units': 'seconds'}
```

CHAPTER TEN

ERROR HANDLING

When handling exceptions and errors in the XML functions, we need to consider what kind of usage is most common and what is required.

There are essentially two sides to this:

- If the functions are used in the context of AiiDA, an exception should be recoverable and as much information should still be used as possible together with logs, where all occuring errors are clearly documented
- If the functions are used in the context of user scripts directly it is much more desirable to except as early as the program knows, that the result is not usable, for example `None`, and let the user make the necessary modifications

To handle both of these cases the standard approach is for functions to provide an optional argument `logger`, which takes a `Logger` from the corresponding library of the stdlib of python. If this argument is provided, we are in the first case and functions should only except for really critical errors and return maybe less usable but no no information.

As an example the evaluation routines will return the unconverted string from the XML file if the `logger` argument is given and no conversion was successful but it could be retrieved from the file. On the other hand if the `logger` argument is not given the function will raise an error. With this behaviour users can be sure that if the function did not raise an error all conversions were successful.

CHAPTER ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search