PyAlaOCL Documentation

Release 0.0.3

megaplanet

March 28, 2015

Contents

PyAlaOCL is a small framework that aims to bring features *à la* OCL to python world. OCL refers to the **Object Constraint Language** of UML (Unified Modeling Language).

PyAlaOCL does *not* provide a OCL interpreter implemented in python. It "just" allows python programmers familiar with OCL to write expressions *à la* OCL in python programs. For instance instead of writing the following OCL expression:

```
Set{-3,2,-1,-2,-5}->excluding(2)->forAll(x|x<0)
```

joe the programmer will write instead this python expression:

```
>>> Set(-3,2,-1,-2,-5).excluding(2).forAll(lambda x:x<0)
True
```

While the OCL syntax is not retained (python syntax can not be extended), the power of OCL is still there. Almost the full OCL library is supported including features like *closure* making it possible to write rather complex traversals in a rather concise and elegant way. Some additional operators are also added to take profit of python while keeping the spirit of OCL.

And last, but not least, PyAlaOCL can (optionally) be integrated in different python settings making it even more handy to use:

- jinja2 integration. PyAlaOCL expressions can be written within jinja2 templates, increasing the expression power of jinja2.
- jython integration. Java collections such as Set or List can be instrumented so that PyAlaOCL expressions work on them. This makes it possible to work with Java apis in a seamless way.
- modelio integration. PyAlaOCL can be used in the context of modelio, the open source UML environment, bringing *à la* OCL support to modelio.
- USE OCL integration. The integration with the USE OCL environment has been moved to its own project. See PyUseOCL documentation.

The code is open source, and available at github.

CHAPTER 1

Documentation

1.1 Quick Start Guide

You should be able to install and try PyAlaOCL in minutes (assuming that you have a decent python environment installed and in particular pip).

1.1.1 Installation

Just like nearly all python packages, PyAlaOCL can be installed using pip:

```
$ pip install pyalaocl
```

You can also download the source from the PyAlaOCL github project.

1.1.2 Giving it a first try

Just launch the python interpreter and try the following statements:

```
C:\joe\> python
Python 2.7.8 ...
Type "help", "copyright", "credits" or "license" for more information.
>>> from pyalaocl import *
>>> Seq("hello","world",2015).selectByKind(str)
Seq(hello, world)
```

1.2 Rationales

The design of PyAlaOCL is due to various rationales.

1.2.1 Why PyAlaOCL?

While the OCL specification corresponds to a rather a cool language, not so many implementation exists. Currently OCL is mostly available as: * standalone tools, such as USE OCL * OCL variants or subsets embedded in high level languages such as model transformation languages such as ATL * OCL interpreters in java environments, most notably OCL Eclipse

We are not aware of any implementation in the context of python. Hence PyAlaOCL...

1.2.2 What is PyAlaOCL?

PyAlaOCL is *not* an interpreter of the OCL language implemented in the python language. It is best described as the implementation of the OCL library for python programmers allowing them to write à la OCL expressions. While the syntax of OCL is not supported, the expressions writen using python syntax are quite close.

1.2.3 Rationales

PyAlaOCL is developed with the following rationales in mind:

- **simplicity** : Developing
- **python integration** : the goal is to provided a
-

1.3 Features

1.3.1 OCL support

Collection of classes and functions to ease the translation of OCL expressions into python.

- = -> ==
- x.isUndefined() -> isUndefined(x)
- x.isUndefined() -> isUndefined(x)

1.3.2 Number

- x.abs() -> abs(x)
- x.min(y) -> min(x,y)
- x.max(y) -> max(x,y)

1.3.3 Integer

- div /
- mod %

1.3.4 Real

- x.floor() -> floor(x)
- x.round() -> round(x)

1.3.5 String

- s1.size() -> len(s1)
- s1.contact(s2) -> s1+s2
- s1.substring(i1,i2) -> s1[i1,i2] TODO: check
- s1.toUpper() -> s1.upper()
- s1.toLower() -> s1.lower()

1.3.6 Boolean

- true -> True
- false -> False
- xor -> ^ but it must be applied between boolean
- implies -> |implies|
- if c then a else b endif -> (a if c else b)

1.3.7 Enumeration

- E::x -> E.x

1.3.8 Collection

- coll C(coll)
- coll->op(...) C(coll).op(...)
- Set{ ... } -> Set(...)
- Bag{ ... } -> Bag(...)
- OrderedSet{ ... } -> OrderedSet(...)
- Sequence{ ... } -> Seq(...)
- Sequence {1..5, 10..20} -> Seq.new(range(1,5)+range(10,20))

1.3.9 UML based features

- oclIsNew -> Not available. Can be use only with post condition
- oclAsType -> Not necessary thanks for dynamic typing in python.

1.3.10 Examples

This modules provides most core features of PyAlaOCL. It defines the various functions and classes that constitutes the OCL library.

`pyalaocl.`**`floor`**`(`*`r`*`)`

Return the largest integer which is not greater than the parameter.

`pyalaocl.isUndefined(value)`

Indicates if the given parameter is undefined (None) or not. :param value: any kind of value. :type value: any :return: True if the value is None. :rtype: bool

Examples:

```
>>> print isUndefined(3)
False
>>> print isUndefined(None)
True
```

`pyalaocl.oclIsKindOf(value1, value2)`

Evaluates to True if the type of the value is *exactly* the type given as a second parameter is an instance of type or one of its subtypes directly or indirectly. Use the method oclIsTypeOf if you want to check if a value is exactly of a given type.

Parameters

- **value** (*Any*) – A scalar value, a collection or an object.
- **aType** (*type*) – The type to check the value against (e.g. int, float, str, unicode, bool or a class)

Returns True if value is compatible with the type aType.

Return type bool

Examples:

```
>>> print oclIsKindOf(3,int)
True
>>> print oclIsKindOf("3",int)
False
>>> print oclIsKindOf(2.5,float)
True
>>> print oclIsKindOf("hello",basestring)
True
>>> print oclIsKindOf(True,bool)
True
>>> class Person(object): pass
>>> print oclIsKindOf(Person(),Person)
True
>>> print oclIsKindOf(Person(),object)
True
>>>
```

`pyalaocl.oclIsTypeOf(value1, value2)`

Return True if the type of the value is *exactly* the type given as a second parameter. This function does not take into account sub-typing relationships. If this is what is intended, use oclIsKindOf instead.

Parameters

- **value** (*Any*) – A scalar value, a collection or an object.
- **aType** (*type*) – The type to check the value against (e.g. int, float, str, unicode, bool or a class)

Returns True if value is compatible with the type aType.

Return type bool

Examples:

```
>>> print oclIsTypeOf("hello",str)
True
>>> print oclIsTypeOf("hello",basestring)
False
>>> print oclIsTypeOf(u"çüabè",unicode)
True
```

class pyalaocl.**Collection**

Base class for OCL collections. Collections are either: * sets (Set), * ordered set (OrderedSet) * bags (Bag), * sequences (Seq)

class pyalaocl.**Set**(**args*)

Set of elements.

This class mimics OCL Sets. Being a set, there are no duplicates and no ordering of elements. By contrast to OCL Sets, here a set can contain any kind of elements at the same time. OCL sets are homogeneous, all elements being of the same type (or at least same supertype).

collectNested(*expression*)

Return a bag of values resulting from the evaluation of the given expression on all elements of the set.

The transformation from this set to a bag is due to the fact that the expression can generate duplicates.

Parameters **expression** (*X -> Y*) – A function returning any kind of value.

Returns The bag of values produced.

Rtype Bag[Y]

Examples:

```
>>> Set(2,3,5,-5).collectNested(lambda e:e*e) == Bag(25,25,4,9)
True
>>> Set(2,3).collectNested(lambda e:Bag(e,e))                    == Bag(Bag(2,2),Bag(3,
True
```

count(*value*)

Return the number of occurrence of the value in the set (0 or 1). :param value: The element to search in the set. :type value: any :return: 1 if the element is in the set, 0 otherwise. :rtype: bool

Examples:

```
>>> Set(1,3,"a").count("3")
0
>>> Set(1,3,3,3).count(3)
1
```

difference(*anyCollection*)

Remove from the set all values in the collection. :param anyCollection: Any collection of values to be discarded from this set. :type anyCollection: collection :return: This set without the values in the collection. :rtype: Set

Examples:

```
>>> Set(1,3,"a").difference([2,3,2,'z']) == Set(1,"a")
True
>>> Set(1,3,3,3).difference(Set(1,3)) == Set()
True
```

```
>>> Set().difference(Set()) == Set()
True
>>> Set(1,3) - [2,3] == Set(1)
True
```

duplicates()
Return always an empty bag for a set

excluding(*value*)
Excludes a value from the set (if there). :param value: The element to add to the set. :type value: any :return: A set including this element. :rtype: Set

Examples:

```
>>> Set(1,3,"a").excluding("3") == Set(1,3,"a")
True
>>> Set(1,3,3,3).excluding(3) == Set(1)
True
```

flatten()
If the set is a set of collections, then return the set-union of all its elements. :return: Set :rtype: Set

Examples:

```
>>> Set(Set(2)).flatten() == Set(2)
True
>>> Set(Set(Set(2)),Set(2)).flatten() == Set(2)
True
>>> Set(Set(2,3),Set(4),Set(),Bag("a"),Bag(2,2)).flatten()                    == Set(2,3
True
```

#>>> Set().flatten() == Set() # True # >>> Set(2,3).flatten() == Set(2,3) # True # >>> Set(2,Set(3),Set(Set(2))).flatten() == Set(2,3) # True

hasDuplicates()
Return always False for sets.

This method is an extension to OCL. It makes is defined on sets just for consistency but is more useful for Bags or Sequences. :return: True :rtype: bool

including(*value*)
Add the element to the set if not already there. :param value: The element to add to the set. :type value: any :return: A set including this element. :rtype: Set

Examples:

```
>>> Set(1,3,"a").including("3") == Set(1,"3",3,"a")
True
>>> Set(1,3,3,3).including(3) == Set(3,1)
True
```

intersection(*anyCollection*)
Retain only elements in the intersection between this set and the given collection.

Parameters **anyCollection** (*collection*) – A collection of values to be added to this set.

Returns A set including all values added plus previous set elements.

Return type Set

Examples:

```
>>> Set(1,3,"a").intersection(["a","a",8]) == Set("a")
True
>>> Set(1,3,3,3).intersection(Set(1,3)) == Set(1,3)
True
>>> Set(2).intersection(Set()) == Set()
True
>>> Set(2) & Set(3,2) == Set(2)
True
```

isEmpty()

Examples:

```
>>> Set().isEmpty()
True
>>> Set(Set()).isEmpty()
False
>>> Set(2,3).isEmpty()
False
```

select(*predicate*)

Retain in the set only the elements satisfying the expression.

Parameters **predicate** – A predicate, that is a function returning a boolean.

Returns The set with only the selected elements.

Rtype Set

Examples:

```
>>> Set(2,3,2.5,-5).select(lambda e:e>2) == Set(3,2.5)
True
>>> Set(Set(1,2,3,4),Set()).select(lambda e:e.size()>3)                    == Set(Set(1
True
```

selectWithCount(*number*)

Return the set if 1 is selected as count. Otherwise return an empty set because there is no duplicated elements in a set. :param number: :type number: :return: :rtype:

size()

Return the size of the set. :return: The size of the set. :rtype: int

Examples:

```
>>> Set(1,4,2,1,1).size()
3
>>> Set().size()
0
```

symmetricDifference(*anyCollection*)

Return the elements that are either in one set but not both sets.

In fact this method accept any collection, but it is first converted to a set.

Parameters **anyCollection** (*collection*) – A collection to make the difference with.

Returns The symmetric difference.

Return type Set

Examples:

```
>>> Set(1,2).symmetricDifference(Set(3,2)) == Set(1,3)
True
>>> Set(Set()).symmetricDifference(Set()) == Set(Set())
True
```

union(*anyCollection*)

Add all elements from the collection given to the set. :param anyCollection: A collection of values to be added to this set. :type anyCollection: collection :return: A set including all values added plus previous set elements. :rtype: Set

Examples:

```
>>> Set(1,3,'a').union([2,3,2]) == Set(1,3,"a",2)
True
>>> Set(1,3,3,3).union(Set(2,1,8)) == Set(1,2,3,8)
True
>>> Set().union(Set()) == Set()
True
>>> Set(1,3) | [2,3] == Set(1,2,3)
True
```

pyalaocl.**asSet**(*collection*)

Convert the given collection to a Set :param collection: :return: :rtype: Set

pyalaocl.**asBag**(*anyCollection*)

Parameters **anyCollection** –

Returns

pyalaocl.**asSeq**(*anyCollection*)

Convert the given collection to a Seq :param anyCollection: :return: :rtype: Seq

pyalaocl.**isCollection**(*value*, *language=None*)

Parameters

- **value** –
- **language** –

Returns

```
>>> isCollection((2,3))
True
>>> isCollection([])
True
>>> isCollection(12)
False
>>> isCollection(Counter())
True
>>> isCollection("text")
False
```

pyalaocl.**asCollection**(*anyCollection*)

Convert any collection into the proper (OCL) collection.

Parameters **anyCollection** – A python, java or ocl collection.

Returns The OCL collection

Return type Collection

Examples:

```
>>> asCollection({2,3}) == Set(3,2)
True
>>> asCollection(frozenset({1,5,1})) == Set(1,5)
True
>>> asCollection(Counter([1,1,3,1])) == Bag(1,1,1,3)
True
>>> asCollection(Counter({'hello':2,-1:0})) == Bag('hello','hello')
True
>>> asCollection([1,2,3,4]) == Seq(1,2,3,4)
True
>>> asCollection((1,2,3,4)) == Seq(1,2,3,4)
True
>>> asCollection(deque([1,2,3,4])) == Seq(1,2,3,4)
True
```

1.4 Limitations

The current implementation of PyAlaOCL suffers from various limitations with respect to the OCL standard. Some limitations will remain in the further as they are linked by to the rationales behind PyAlaOCL. Other limitations may disappear in the future with further development effort.

1.4.1 Significant limitations

- **No support for OCL syntax**. Python syntax has to be used instead. For instance instead of writing -> for operators one collections one have to write ..
- **No implicit context binding**. While in OCL one can write expression like *people->forAll(x > 0)* in PyAlaOCL one have to write *people.forAll(lambda p:p.x)* if x is an attribute of the Person type.
- The scope of all iterate operations is limited to the variable of the previous iteration and no other variable coming from the context are used.
- Operations cannot be applied to python built-in collections such as tuple, list and set.

1.4.2 Not implemented yet

The following OCL features are not implemented yet:

- *OrderedSet* type.
- *iterate* operation on collections.
- *Tuple* type.
- Product operation.
- Multiple variables in *forAll* and *exists*.

1.5 Jinja2 Integration

The pyalaocl.jinja2 module provides the necessary features to use pyalaocl expressions into jinja2 templates. To be more precise a function is provided to augment jinja2 environment with:

- jinja filters. asSet, asBag, asSeq can be used as filters.
- jinja global symbols.

In the context of Modelio, global functions are also added as global symbols.

`pyalaocl.jinja2.`**`addOCLToEnvironment`**(*jinja2Environment*)

Add OCL functions to a jinja2 environment so that OCL can be used in jinja2 templates.

Parameters **jinja2Environment** (*jinja2.Environment*) – Jinja2 environment to be instrumented.

Returns The modified environment.

Return type jinja2.Environment

1.6 Jython Integration

1.7 Modelio Integration

PyAlaOCL expressions can be used in the jython interpreter of the modelio open source environment, making it possible to evaluate OCL expressions on UML or BPMN models.

1.7.1 Rationales

While modelio open source tool is an excellent tool it does not currently provide OCL support. Its scripting environment based on jython is however one of its best features.

1.7.2 pyalaocl.modelio

1.7.3 pyalaocl.modelio.profiles

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Index