pyxtuml Documentation

Release 1.0.0

John Törnblom

Jul 23, 2018

Contents

Relational information modeling in Python using BridgePoint.

Contents:

CHAPTER 1

Getting Started

1.1 Dependencies

In addition to python itself, pyxtuml also depend on the python library ply. For people running Ubuntu, everything is available via apt-get:

```
$ sudo apt-get install python2.7 python-ply
```

pyxtuml also works with python3 and pypy.

1.2 Installation

pyxtuml is published on pypi, and thus may be installed using pip:

```
$ python -m pip install pyxtuml
```

You could also fetch the source code from github and install it manually:

```
$ git clone https://github.com/xtuml/pyxtuml.git
$ cd pyxtuml
$ python setup.py install
```

Optionally, you can also execute a test suite:

```
$ python setup.py test
```

1.3 Usage example

The examples folder contains a few scripts which demonstrate how pyxtuml may be used.

The following command will create an empty metamodel and populate it with some sample data:

```
$ python examples/create_external_entity.py > test.sql
```

Copy the SQL statements saved in test.sql to your clipboard, and paste them into the BridgePoint editor with a project selected in the project explorer.

If you are on a more recent GNU/Linux system, you can also pipe the output directly to your clipboard without bouncing via disk:

```
$ python examples/create_external_entity.py | xclip -selection clipboard
```

CHAPTER 2

Command Line Tools

pyxtuml contain a few useful command line tools which are described below.

2.1 Consistency Check

A model may be checked for association constraint violations. By default, all associations present in a model are checked. Optionally, the check may be limited to one or more associations by appending the -r argument for each association to check.

```
$ python -m xtuml.consistency_check [options] <sql_file> [another_sql_file...]
```

Note: both the model and its schema needs to be provided by the user.

Available options

Option	Description
–version	show program's version number and exit
–help, -h	show this help message and exit
-r <number>	limit consistency check to one or more associations
-k <key letter>	limit check for uniqueness constraint violations to one or more classes
–verbosity, -v	increase debug logging level

2.1.1 BridgePoint metamodel

There is also a tool available that checks for constraint violations in ooaofooa, the metamodel used by the BridgePoint editor. It can be used to detect various fatal issues in a BridgePoint model, e.g. parameters that lacks a type.

```
$ python -m bridgepoint.consistency_check [options] <model_path> [another_model_path..
↪.]
```

Available options

Option	Description
–version	show program's version number and exit
–help, -h	show this help message and exit
-r <number>	limit consistency check to one or more associations
-k <key letter>	limit check for uniqueness constraint violations to one or more classes
–globals, -g	add builtin global data types automatically, e.g. boolean, integer and real
–verbosity, -v	increase debug logging level

2.2 SQL Schema Generator

To create an sql schema from a BridgePoint model, the following command may be used:

```
$ python -m bridgepoint.gen_sql_schema [options] <model_path> [another_model_path...]
```

Available options

Option	Description
–version	show program's version number and exit
–help, -h	show this help message and exit
–component=NAME, -c NAME	export sql schema for the component named NAME
–derived-attributes, -d	include derived attributes in the schema
–output=PATH, -o PATH	save sql schema to PATH (required)
–verbosity, -v	increase debug logging level

2.3 XSD Schema Generator

To create an XSD schema for XML files, the following command may be used:

```
$ python -m bridgepoint.gen_xsd_schema [options] <model_path> [another_model_path...]
```

Available options

Option	Description
–version	show program's version number and exit
–help, -h	show this help message and exit
–component=NAME, -c NAME	export xsd schema for the component named NAME
–output=PATH, -o PATH	save xsd schema to PATH (required)
–verbosity, -v	increase debug logging level

Note that the XSD schema is compatible with Microsoft Excel. Consequently, Excel may be used to define instances in a model that can be easily exported to XML files.

2.4 OAL Prebuilder

Generally, all model compilers takes as input an sql where all OAL actions has been translated from its textual representation into instances in the ooaofooa meta model. This translation is usually conducted by the Eclipse- based

prebuilder included with the BridgePoint IDE. pyxtuml contains an independent prebuilder, implemented in python (and thus may be somewhat slower). The pyxtuml prebuilder may be invoked using the folling command:

```
$ python -m bridgepoint.prebuild [options] <model_path> [another_model_path..]
```

Available options

Option	Description
–version	show program's version number and exit
–help, -h	show this help message and exit
–verbosity, -v	increase debug logging level
–output=PATH, -o PATH	set output to PATH

CHAPTER 3

API Reference

3.1 xtuml

The following section lists functions, classes and exceptions from the xtuml module. The operations are independent of the underlying metamodel definition, i.e. the sql schema.

3.1.1 Loading Metamodels

xtuml.**load_metamodel**(*resource*)

Load and return a metamodel from a *resource*. The *resource* may be either a filename, or a list of filenames.

Usage example:

```
>>> metamodel = xtuml.load_metamodel(['schema.sql', 'data.sql'])
```

class xtuml.**ModelLoader**

Class for loading metamodels previously persisted to disk.

Data may be provided in any order, e.g. instances followed by associations, followed by class definitions. One single loader may be used to build several *xtuml.MetaModel* objects, and additional data may be provided at any time.

Note: Additional data will not affect previosly built metamodels.

Usage example:

```
>>> l = xtuml.ModelLoader()
>>> l.filename_input('data.sql')
>>> l.filename_input('schema.sql')
>>> m1 = l.build_metamodel()
>>> l.filename_input('additional_data.sql')
>>> m2 = l.build_metamodel()
```

build_metamodel (*id_generator=None*)
Build and return a *xtuml.MetaModel* containing previously loaded input.

file_input (*file_object*)
Read and parse data from a *file object*, i.e. the type of object returned by the builtin python function *open()*.

filename_input (*filename*)
Open and read from a *filename* on disk, and parse its content.

input (*data*, *name='<string>'*)
Parse *data* directly from a string. The *name* is used when reporting positional information if the parser encounter syntax errors.

populate (*metamodel*)
Populate a *metamodel* with entities previously encountered from input.

3.1.2 Metamodel Operations

class xtuml.**MetaModel** (*id_generator=None*)
A metamodel contains metaclasses with associations between them.

Note: All identifiers, e.g. attributes, association ids, key letters (the kind or name of a class), are case **insensitive**.

clone (*instance*)
Create a shallow clone of an *instance*.

Note: the clone and the original instance **does not** have to be part of the same metaclass.

find_class (*kind*)
Find a class of some *kind* in the metamodel.

new (*kind*, **args*, ***kwargs*)
Create and return a new instance in the metamodel of some *kind*.

Optionally, initial attribute values may be assigned to the new instance by passing them as positional or keyword arguments. Positional arguments are assigned in the order in which they appear in the metaclass.

select_many (*kind*, *where_clause=None*)
Query the metamodel for a set of instances of some *kind*. Optionally, a conditional *where-clause* in the form of a function may be provided.

Usage example:

```
>>> m = xtuml.load_metamodel('db.sql')
>>> inst_set = m.select_many('My_Class', lambda sel: sel.number > 5)
```

select_one (*kind*, *where_clause=None*)
Query the metamodel for a single instance of some *kind*. Optionally, a conditional *where-clause* in the form of a function may be provided.

Usage example:

```
>>> m = xtuml.load_metamodel('db.sql')
>>> inst = m.select_one('My_Class', lambda sel: sel.name == 'Test')
```

xtuml.**navigate_one** (*instance*)
Initialize a navigation from one *instance* to another across a one-to-one association.

The resulting query will return an instance or None.

Usage example:

```
>>> from xtuml import navigate_one as one
>>> m = xtuml.load_metamodel('db.sql')
>>> inst = m.select_any('My_Modeled_Class')
>>> other_inst = one(inst).Some_Other_Class[4]()
```

The syntax is somewhat similar to the action language used in BridgePoint. The same semantics would be expressed in BridgePoint as:

```
select any inst from instances of My_Modeled_Class;
select one other_inst related by inst->Some_Other_Class[R4];
```

Note: If the navigated association is reflexive, a phrase must be provided, e.g.

```
>>> other_inst = one(inst).Some_Other_Class[4, 'some phrase']()
```

xtuml.**navigate_any**(*instance_or_set*)
Initialize a navigation from an instance, or a set of instances, to associated instances across a one-to-many or many-to-many association.

The resulting query will return an instance or None.

xtuml.**navigate_many**(*instance_or_set*)
Initialize a navigation from an instance, or a set of instances, to associated instances across a one-to-many or many-to-many association.

The resulting query will return a set of instances.

xtuml.**navigate_subtype**(*supertype*, *rel_id*)
Perform a navigation from *supertype* to its subtype across *rel_id*. The navigated association must be modeled as a subtype-supertype association.

The return value will an instance or None.

xtuml.**relate**(*from_instance*, *to_instance*, *rel_id*, *phrase=''*)
Relate *from_instance* to *to_instance* across *rel_id*. For reflexive association, a *phrase* indicating the direction must also be provided.

The two instances are related to each other by copying the identifying attributes from the instance on the TO side of a association to the instance n the FROM side. Updated values which affect existing associations are propagated. A set of all affected instances will be returned.

xtuml.**unrelate**(*from_instance*, *to_instance*, *rel_id*, *phrase=''*)
Unrelate *from_instance* from *to_instance* across *rel_id*. For reflexive associations, a *phrase* indicating the direction must also be provided.

The two instances are unrelated from each other by reseting the identifying attributes on the FROM side of the association. Updated values which affect existing associations are propagated. A set of all affected instances will be returned.

xtuml.**delete**(*instance*)
Delete an *instance* from its metaclass instance pool.

xtuml.**where_eq**(***kwargs*)
Return a where-clause that filters out instances based on named keywords.

Usage example:

```
>>> from xtuml import where_eq as where
>>> m = xtuml.load_metamodel('db.sql')
>>> inst = m.select_any('My_Modeled_Class', where(My_Number=5))
```

xtuml.**sort_reflexive**(*set_of_instances*, *rel_id*, *phrase*)
Sort a *set of instances* in the order they appear across a conditional and reflexive association. The first instance in the resulting ordered set is **not** associated to an instance across the given *phrase*.

class xtuml.**MetaClass**(*kind*, *metamodel=None*)
A metaclass contain metadata for instances, e.g. what attributes are available, what thier types are, and so on.

In addition, each metaclass also handle allocations of instances.

append_attribute(*name*, *type_name*)
Append an attribute with a given *name* and *type name* at the end of the list of attributes.

attribute_names
Obtain an ordered list of all attribute names.

clone(*instance*)
Create a shallow clone of an *instance*.

Note: the clone and the original instance **does not** have to be part of the same metaclass.

delete(*instance*)
Delete an *instance* from the instance pool. If the *instance* is not part of the metaclass, a *MetaException* is thrown.

delete_attribute(*name*)
Delete an attribute with a given *name* from the list of attributes.

insert_attribute(*index*, *name*, *type_name*)
Insert an attribute with a given *name* and *type name* at some *index* in the list of attributes.

navigate(*inst*, *kind*, *rel_id*, *phrase=''*)
Navigate across a link with some *rel_id* and *phrase* that yields instances of some *kind*.

new(**args*, ***kwargs*)
Create and return a new instance.

query(*dictonary_of_values*)
Query the instance pool for instances with attributes that match a given *dictonary of values*.

select_many(*where_clause=None*)
Select several instances from the instance pool. Optionally, a conditional *where-clause* in the form of a function may be provided.

select_one(*where_clause=None*)
Select a single instance from the instance pool. Optionally, a conditional *where-clause* in the form of a function may be provided.

xtuml.**check_association_integrity**(*m*, *rel_id=None*)
Check the model for integrity violations on association(s).

xtuml.**check_uniqueness_constraint**(*m*, *kind=None*)
Check the model for uniqueness constraint violations.

3.1.3 Persistance

xtuml.**persist_database**(*metamodel*, *path*)
Persist all instances, class definitions and association definitions in a *metamodel* by serializing them and saving

to a *path* on disk.

xtuml.**persist_instances**(*metamodel*, *path*)
Persist all instances in a *metamodel* by serializing them and saving to a *path* on disk.

xtuml.**persist_schema**(*metamodel*, *path*)
Persist all class and association definitions in a *metamodel* by serializing them and saving to a *path* on disk.

xtuml.**serialize**(*resource*)
Serialize some xtuml *resource*, e.g. an instance or a complete metamodel.

xtuml.**serialize_database**(*metamodel*)
Serialize all instances, class definitions, association definitions, and unique identifiers in a *metamodel*.

xtuml.**serialize_schema**(*metamodel*)
Serialize all class and association definitions in a *metamodel*.

xtuml.**serialize_instances**(*metamodel*)
Serialize all instances in a *metamodel*.

xtuml.**serialize_instance**(*instance*)
Serialize an *instance* from a metamodel.

3.1.4 Tools

class xtuml.**UUIDGenerator**
A uuid-based id generator. 128-bit unique numbers are generated randomly each time a new id is requested.

class xtuml.**IntegerGenerator**
An integer-based id generator. Integers are generated sequentially, starting from the number one.

Generally, the uuid-based id generator shall be used. In some cases such as testing however, having deterministic unique ids may be benifitial.

Usage example:

```
>>> l = xtuml.ModelLoader()
>>> l.filename_input("schema.sql")
>>> l.filename_input("data.sql")
>>> m = l.build_metamodel(xtuml.IntegerGenerator())
```

class xtuml.**Walker**
A walker may be used to walk a tree.

visitors = None

accept(*node*, ***kwargs*)
Invoke the visitors before and after decending down the tree. The walker will also try to invoke a method matching the pattern *accept_<type name>*, where <type name> is the name of the accepted *node*.

default_accept(*node*, ***kwargs*)
The default accept behaviour is to decend into the iterable member *node.children* (if available).

class xtuml.**Visitor**
A visitor may be used to visit tree nodes walked by a walker.

default_enter(*node*)
The default behaviour when entering a *node* if no other action is defined by a subclass is to do nothing.

default_leave(*node*)
The default behaviour when leaving a *node* if no other action is defined by a subclass is to do nothing.

enter(*node*)
Tries to invoke a method matching the pattern *enter_<type name>*, where <type name> is the name of the type of the *node*.

leave(*node*)
Tries to invoke a method matching the pattern *leave_<type name>*, where <type name> is the name of the type of the *node*.

class xtuml.**NodePrintVisitor**
A visitor that prints a tree-like structure to stdout.

default_render(*node*)
The default behaviour when rendering a *node* if no other rendering method is defined by a subclass is to render the class name.

render(*node*)
Try to invoke a method matching the pattern *render_<type name>*, where <type name> is the name of the rendering *node*.

3.1.5 Exceptions

exception xtuml.**ParsingException**
An exception that may be thrown while loading (and parsing) a metamodel.

exception xtuml.**MetaException**
Base class for all exceptions thrown by the xtuml.meta module.

exception xtuml.**DeleteException**
An exception that may be thrown during delete operations.

exception xtuml.**RelateException**
An exception that may be thrown during relate operations.

exception xtuml.**UnrelateException**
An exception that may be thrown during unrelate operations.

exception xtuml.**MetaModelException**
Base class for exceptions thrown by the MetaModel class.

exception xtuml.**UnknownLinkException**(*from_kind*, *to_kind*, *rel_id*, *phrase*)
An exception that may be thrown when a link is not found.

exception xtuml.**UnknownClassException**
An exception that may be thrown when a metaclass is not found.

3.2 bridgepoint

The following section lists functions and classes from the bridgepoint module. All operations require input expressed in the BridgePoint metamodel (ooaofooa).

3.2.1 Loading Models

bridgepoint.**load_metamodel**(*resource=None*, *load_globals=True*)
Load and return a metamodel expressed in ooaofooa from a *resource*. The resource may be either a filename, a path, or a list of filenames and/or paths.

class bridgepoint.**ModelLoader**(*load_globals=True*)
A *xtuml.MetaModel* loader with ooaofooa schema and globals pre-defined.

build_component(*name=None*, *derived_attributes=False*)
Instantiate and build a component from ooaofooa named *name* as a pyxtuml model. Classes, associations, attributes and unique identifers, i.e. O_OBJ, R_REL, O_ATTR in ooaofooa, are defined in the resulting pyxtuml model.

Optionally, control whether *derived attributes* shall be mapped into the resulting pyxtuml model as attributes or not.

Futhermore, if no *name* is provided, the entire content of the ooaofooa model is instantiated into the pyxtuml model.

filename_input(*path_or_filename*)
Open and read input from a *path or filename*, and parse its content.

If the filename is a directory, files that ends with .xtuml located somewhere in the directory or sub directories will be loaded as well.

3.2.2 Model Transformation

bridgepoint.**gen_text_action**(*instance*)
Generate textual OAL action code from an *instance* in the BridgePoint metamodel. The input may be an instance of the following classes:

- S_SYNC
- S_BRG
- O_TFR
- O_DBATTR
- SM_ACT
- SPR_RO
- SPR_RS
- SPR_PO
- SPR_PS

In addition, anything in the ooaofooa subsystems Value or Body, e.g. ACT_SMT or V_VAL are also supported.

bridgepoint.**prebuild_action**(*instance*)
Transform textual OAL actions of an *instance* to instances in the ooaofooa subsystems Value and Body. The provided *instance* must be an instance of one of the following classes:

- S_SYNC
- S_BRG
- O_TFR
- O_DBATTR
- SM_ACT
- SPR_RO
- SPR_RS
- SPR_PO

- SPR_PS

bridgepoint.**prebuild_model**(*metamodel*)

Transform textual OAL actions in a ooaofooa *metamodel* to instances in the subsystems Value and Body. Instances of the following classes are supported:

- S_SYNC
- S_BRG
- O_TFR
- O_DBATTR
- SM_ACT
- SPR_RO
- SPR_RS
- SPR_PO
- SPR_PS

CHAPTER 4

Contributing

If you encounter problems with pyxtuml, please file a github issue. If you plan on sending pull request which affect more than a few lines of code, please file an issue before you start to work on you changes. This will allow us to discuss the solution properly before you commit time and effort.

CHAPTER 5

License

pyxtuml is licensed under the LGPLv3+.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

R

S

U

V

W