UGE Python Configuration Library: High-Level Design

August 31, 2016

Prepared by:

Siniša Veseli
Consultant
Skaisoft Corp.
sinisa.veseli@skaisoft.com

Change History

Date	Description	Author
4/18/16	Initial version.	S. Veseli
8/31/16	Revised document to reflect changes after implementation.	S. Veseli

Table of Contents

1 Introduction

The purpose of this document is to describe high-level design for the Univa Grid Engine Python Configuration Library (UGE PyCL) that can be used by other applications and tools to configure and manage one or more UGE clusters.

2 Assumptions and Remarks

- UGE PyCL will require access to UGE qmaster via the “qconf” command.
- There will be no user authentication/authorization performed by the library. All library calls utilize the same user account under which they are invoked. Hence, any library calls that modify UGE configuration will have to be invoked from an account that has appropriate UGE administrative privileges.
- Ideas and design concepts discussed in this document do not rely on a specific Python version. However, some examples and code snippets may use modules or syntax specific to Python 2.
- UGE software version used for developing this design was 8.1.3p7 [1].
- This document attempts to be as detailed as possible in terms of specifying UGE PyCL high-level design and functionality. Nevertheless, it is expected that the library implementation phase will likely result in additions or slight modifications of some features.

3 High-Level Design

In this section we describe considerations driving UGE PyCL design.

3.1 Communication with Qmaster

UGE PyCL will communicate with qmaster via the “qconf” command: retrieving UGE objects will involve parsing output of qconf “display” commands, while adding or modifying objects will require constructing appropriate object definition files and passing those to qconf add or modify commands. Hence, the high-level library API class will need the same set of variables as the qconf command itself:

- SGE_ROOT
- SGE_CELL
- SGE_QMASTER_PORT
- SGE_EXECD_PORT

The above variables can be either passed to the API class constructor, or they can be inherited from the user’s environment. Note that configuring API objects programmatically should allow different API class instances to manage different UGE clusters at the same time. For example,

one should be able to copy queue configuration from one cluster to another using the following Python script:

```
qconf1 = QconfApi(sge_root='/opt/uge', sge_cell='default',
                  sge_qmaster_port=11111, sge_execd_port=11112)
qconf2 = QconfApi(sge_root='/opt/uge2', sge_cell='default',
                  sge_qmaster_port=21111, sge_execd_port=21112)
all_q = qconf1.get_queue('all.q')
qconf2.modify_queue(all_q)
```

Figure 1: Python code snippet that illustrates using multiple API class instances to manage different clusters.

Note that various PyCL objects and API methods will be described in more detail later in this document.

3.2 UGE Objects and their Python Representation

Although their formatting and display are not consistent across different qconf commands, all UGE objects fit into one of the following three categories:

- Collection of key/value pairs (queues, host groups, projects, job classes, etc.)
- List of strings (lists of manager names, queue names, execution host names, etc.)
- List of key/value collections (share tree, list of resource quota sets)

3.2.1 Collections of Key/Value Pairs

All individual UGE objects with more than one attribute can be described as collections of key/value pairs. A sample definition of project "P1" illustrates this in Figure 2:

```
$ qconf -sprj P1
name P1
oticket 100
fshare 100
acl ACL1
xacl NONE
```

Figure 2: Sample UGE project "P1" object, shown using output of the "qconf -sprj P1" command.

In Python, the natural representation for all such objects is dictionary, which is easily converted into JSON, as illustrated in Figure 3.

```
>>> import json
>>> p1_data = {'oticket': 100, 'xacl': 'NONE', 'fshare': 100, 'name': 'P1', 'acl':
'ACL1'}
>>> json.dumps(p1_data)
'{"oticket": 100, "xacl": "NONE", "acl": "ACL1", "fshare": 100, "name": "P1"}'
```

Figure 3: Python code snippet showing dictionary that represents sample UGE project "P1", as well as its conversion to the JSON string.

In most cases we can simply use native python types or strings as containers for values assigned to UGE object keys (UGE keywords like 'NONE' will be discussed later). There are,

however, some cases where using a list of strings or a dictionary works slightly better in terms of parsing or modifying key values. For example, values for several “queue” object keys like “hostlist” and “slots” can be represented using lists of strings, while values for all keys of the “complex” object can be represented using dictionaries (see Figure 4).

```
>>> complex_data = {'arch' : {'shortcut' : 'a', 'type' : 'RESTRING',
'relop' : '==', 'requestable' : 'YES', 'consumable' : 'NO', 'default' :
'NONE', 'urgency' : 10, 'aapre' : 'NO'}}
```

Figure 4: Python code snippet showing a dictionary representation of the “arch” key value in the UGE “complex” configuration.

3.2.2 List of Strings

Python list of strings is a natural representation qconf commands that return list of names. An example is a list of execution hosts (see Figures 5 and 6).

```
$ qconf -sel
uge-exec-001
uge-exec-002
uge-exec-003
```

Figure 5: Sample output of the “qconf -sel” command.

```
>>> import json
>>> host_list = ['uge-exec-001', 'uge-exec-002', 'uge-exec-003']
>>> json.dumps(host_list)
'["uge-exec-001", "uge-exec-002", "uge-exec-003"]'
```

Figure 6: Python code snippet showing list of execution hosts, as well as its conversion to the JSON string.

3.2.3 List of Key/Value Collections

For those qconf commands that return list of key/value collections we can use Python list of dictionaries (see Figures 7 and 8 showing a sample share tree and its Python representation).

```
id=0
name=Root
type=0
shares=1
childnodes=1,2
id=1
name=U1
type=0
shares=10
childnodes=NONE
id=2
name=U2
type=0
shares=25
childnodes=NONE
```

Figure 7: Sample output of the “qconf -sstree” command.

```
>>> import json
>>> stree_data = [{'id' : 0, 'name' : 'Root', 'type' : 0, 'shares' : 1,
'childnodes' : [1,2]}, {'id' : 1, 'name' : 'U1', 'type' : 0, 'shares' : 10,
'childnodes' : 'NONE'}, {'id' : 2, 'name' : 'U2', 'type' : 0, 'shares' : 25,
'childnodes' : 'NONE'}]
>>> json.dumps(stree_data)
'[{"childnodes": [1, 2], "type": 0, "id": 0, "shares": 1, "name": "Root"},
{"childnodes": "NONE", "type": 0, "id": 1, "shares": 10, "name": "U1"},
{"childnodes": "NONE", "type": 0, "id": 2, "shares": 25, "name": "U2"}]'
```

Figure 8: Python code snippet showing sample share tree representation, as well as its conversion to the JSON string.

3.3 Metadata versus UGE Data

Effective configuration management of multiple UGE clusters requires information about various UGE objects, in addition to actual object configuration data that was discussed in Section 3.2. For example, UGE objects may evolve between different software versions, and hence it is imperative to know which object version was actually stored in the configuration database. Another example is tracking configuration changes: cluster administrators might like to know the time/date of the most recent configuration change, which administrator made the change, and why.

For this reason, PyCL objects will contain both UGE configuration data and object metadata. Different configuration tools might rely on different sets of metadata keys, and the list of metadata keys in usage will likely grow with time. At a minimum, however, metadata will have to contain set of static keys that are needed by the PyCL library itself (e.g., for recreating configuration objects from JSON strings):

- object_class
- object_version

In addition to those, some of the optional metadata keys that may be added to PyCL objects are listed below:

- object_name (may be needed for identifying some objects)
- created_by
- created_on
- modified_by
- modified_on
- uge_cluster
- uge_version
- description
- …

Note that some of the above metadata keys may be added automatically by the PyCL library, while others may be added by configuration tools using PyCL object interfaces.

Although PyCL objects may have other data members, their JSON representations will be based on dictionary formed only from metadata dictionary key/value pairs, as well as from UGE object assigned to the reserved key “data”. For example, JSON string representation for a PyCL object that “wraps” sample UGE project “P1” from Figure 2 is shown below:

```
>>> import json
>>> p1_data = {'oticket': 100, 'xacl': 'NONE', 'fshare': 100, 'name': 'P1',
'acl': 'ACL1'}
>>> p1_json_data = {'object_class' : 'Project', 'object_version' : '1.0',
'modified_by' : 'sveseli', 'modified_on' : '2016-04-15T19:30:07.969048',
'uge_cluster' : 'C1', 'data' : p1_data}
>>> json.dumps(p1_json_data)
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"data": {"oticket": 100, "xacl": "NONE", "acl": "ACL1", "fshare": 100, "name":
"P1"}, "modified_on": "2016-04-15T19:30:07.969048", "object_class": "Project"}'
```

Figure 9: Python code snippet showing full JSON string representation (data and metadata) for a sample UGE “P1” project.

Note that timestamps in metadata will be strings using ISO 8601 format.

3.4 PyCL Object Classes

All qconf definitions and logic related to a given UGE object will be encapsulated within its corresponding PyCL wrapper class. For example, the PyCL “Queue” class will know that retrieving UGE queue objects requires a “`qconf -sq <queue_name>`” command, it will have the list of keys needed to create a new queue object, it will provide interfaces needed to modify queue configuration, it will know how to parse and unpack JSON string, and also how to prepare files suitable for a “`qconf -Aq <queue_definition_file>`” or for a “`qconf -Mq <queue_definition_file>`” command.

3.4.1 QconfObject Class

PyCL object classes will derive from a common “QconfObject” base class (see Figure 10). This class will contain functionality and interfaces common to all PyCL objects (e.g., conversion to JSON strings, logic for parsing output of qconf commands that display UGE objects, etc.).

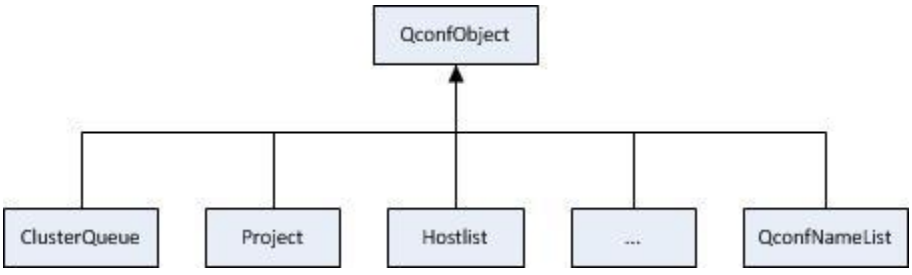

Figure 10: All PyCL object classes will use QconfObject as a base class.

In addition to classes wrapping UGE objects, the above diagram also contains special “QconfNameList” class, which will support all regular Python list features, and will be used as a container for various lists of object names, such as cluster queues, managers, execution hosts, etc.

3.4.2 Initializing Objects

Constructors for QconfObject-based classes will take several named (and all optional) arguments: “data”, “metadata”, and “json” (see Figure 11).

```
class QconfObject:
    def __init__(self, name=None, data=None, metadata=None, json=None):
        ...
```

Figure 11: QconfObject class constructor signature.

The “data” argument should contain UGE object configuration data (e.g., dictionary in case of objects like queues or projects, list of dictionaries for share trees). The “metadata” argument should be a dictionary containing any user-provided key/value pairs, which will be merged with static class metadata. The “json” argument should contain object’s JSON string representation discussed earlier. If specified, JSON string will be unpacked, and resulting key/value pairs will be merged with the provided metadata and/or data. In case identical keys are specified in multiple places (e.g., in both metadata dictionary and as part of the JSON string), values coming from the JSON string will take precedence.

The scheme described above allows for creating empty, partially or fully configured objects, and also objects that can be easily updated and manipulated, as shown in Figure 12.

```
>>> all_q = ClusterQueue()
>>> all_q.data['qname'] = 'all.q'
```

Figure 12: Python code snippet that shows how an empty cluster queue object can be created and manipulated.

Keep in mind, however, that requirement for supporting multiple UGE releases complicates things in terms of invoking object constructors using classes that are applicable for a given UGE release. For this reason, the PyCL library will provide object factory methods that should be able to instantiate appropriate versions of object wrapper classes, using either provided JSON string representation, or UGE object data (see Figure 13).

```
qconf = QconfApi(sge_root='/opt/uge', sge_cell='default')
all_q = qconf.generate_queue(data={'qname' : 'all.q'})
```

Figure 13: Python code snippet showing sample usage of factory methods for creating PyCL objects.

3.4.3 Support for API CRUD Operations

In order to support API Create, Read, Update and Delete (CRUD) operations PyCL UGE object wrapper classes will have to know appropriate set of qconf command switches. Read operations will also require ability to parse qconf “display” command output, and to generate object’s JSON representation. On the other hand, create and update (or modify) operations will need functionality for parsing JSON strings and creating UGE object configuration files.

Most UGE objects have a set of keys that must be present in their configuration files in order for those files to be accepted by the qmaster. For this reason, every PyCL wrapper class will have a dictionary of required keys and their respective default values (see example below for UGE project objects).

```
REQUIRED_DATA_DEFAULTS = {
    'name' : None,
    'oticket' : 0,
    'fshare' : 0,
    'acl' : None,
    'xacl' : None
}
```

Figure 14: Python code snippet showing default values for required data keys for UGE projects.

Most keys that have default values will not have to be provided by the library user. For certain keys values will have to be set by the library user before UGE object is added or modified via the relevant PyCL API calls (e.g., “qname” attribute of UGE cluster queue). Note that default values for certain keys may be installation specific, and will have to be generated at runtime (for example, those keys that rely on $SGE_ROOT)

Some objects (e.g., cluster configuration) have a set of keys that are optional (i.e., they can be removed deleted from object’s configuration). In order to accommodate such cases, PyCL wrapper class may utilize dictionaries of optional data keys and their respective default values. For example:

```
OPTIONAL_HOST_DATA_DEFAULTS = {
    'mailer' : '/bin/mail',
    'xterm' : '/usr/bin/xterm'
}
OPTIONAL_GLOBAL_DATA_DEFAULTS = {
    'execd_spool_dir' : 'SGE_ROOT/SGE_CELL/spool',
    'mailer' : '/bin/mail',
    'xterm' : '/usr/bin/xterm',
    …
}
```

Figure 15: Python code snippet showing default values for optional data keys for UGE cluster (global and host-specific) configuration. Keys that rely on variables like SGE_ROOT and SGE_CELL will have to be evaluated at runtime.

3.4.4 Object Versioning

Over time, UGE objects will evolve with new UGE releases, and hence their corresponding PyCL wrapper classes will have to change as well. Any time PyCL object wrapper class gets modified in a non-trivial manner it will get assigned a new version string. For example, this would happen if a new key gets added to the set of required object keys, or if a default key value changes. The library must keep track of old PyCL object versions as long as the corresponding UGE product release is supported. This means two things:

1) For each UGE object there may more than one version of PyCL wrapper class that must be supported. For example, there may be three supported versions of the Queue class, but only one supported version of the Project class.
2) Each supported UGE release may correspond to a distinct set of PyCL wrapper classes. For example, future UGE versions 9.0 and 10.0 might use Queue class versions 1.0 and 2.0, respectively, and use the same 1.0 version of the Project class.

The first statement above suggests that PyCL could adopt Java-like "class per file" convention, meaning that the definition of each wrapper class will be specified in its own file, with a version in its name. The exact wrapper class file naming convention is not that important, as long as it is applied consistently across all objects. For example, ClusterQueue class versions 1.0 and 2.0 could be specified in files "cluster_queue_v1_0.py" and "cluster_queue_v2_0.py", allowing their usage as in the example below:

```
>>> from cluster_queue_v1_0 import ClusterQueue as ClusterQueueV1
>>> from cluster_queue_v2_0 import ClusterQueue as ClusterQueueV2
>>> q1 = ClusterQueueV1()
>>> q2 = ClusterQueueV2()
>>> print q1.__class__.__name__,q1.VERSION
ClusterQueue 1.0
>>> print q2.__class__.__name__,q2.VERSION
ClusterQueue 2.0
```

Figure 16: Python code snippet showing possible solution for a problem of supporting multiple versions for a given object. In the above example, both imported python modules contain definitions for the class "ClusterQueue" with the same constructor signature, but with a different value specified for the "VERSION" constant.

Support for distinct sets of objects for different UGE releases can be implemented using release/object set map, where release versions serve as keys and dictionary of object names/versions as values (see Figure 17). Such scheme, combined with the above solution for supporting different object versions, would require minimal amount of maintenance and development effort, and would provide adequate support for supporting multiple UGE releases, as well as support for product upgrades.

```
>>> UGE_RELEASE_OBJECT_MAP = {}
>>> UGE_RELEASE_OBJECT_MAP['8.5.0p1'] = { 'ClusterQueue' : '1.0', 'Project' :
'1.0', 'ShareTree' : '1.0' }
>>> UGE_RELEASE_OBJECT_MAP['8.5.0p2'] = { 'ClusterQueue' : '2.0', 'Project' :
'1.0', 'ShareTree' : '1.0' }
>>> UGE_RELEASE_OBJECT_MAP['8.6.0'] = UGE_RELEASE_OBJECT_MAP['8.5.0p2']
>>> UGE_RELEASE_OBJECT_MAP
{'8.6.0': {'ClusterQueue': '2.0', 'Project': '1.0', 'ShareTree': '1.0'},
'8.5.0p2': {'ClusterQueue': '2.0', 'Project': '1.0', 'ShareTree': '1.0'},
'8.5.0p1': {'ClusterQueue': '1.0', 'Project': '1.0', 'ShareTree': '1.0'}}
```

Figure 17: Python code snippet showing construction of UGE_RELEASE_OBJECT_MAP, which can be used for tracking versions of PyCL object classes that are applicable for a given UGE release.

In order to simplify object management, PyCL will use QconfObjectFactory class, which will be able to generate PyCL objects appropriate for any version of UGE software.

3.5 API Functionality

3.5.1 Initialization

As mentioned earlier, QconfApi class will require several variables for communication with qmaster: UGE root directory, cell name, and qmaster and execd ports:

```
qconf = QconfApi(sge_root='/opt/uge', sge_cell='default',
                 sge_qmaster_port=11111, sge_execd_port=11112)
```

Figure 18: Example of QconfApi class initialization.

During initialization QconfApi instance will determine administrator's username, hostname, as well as UGE software version, which will facilitate using PyCL objects appropriate for the given qmaster version (see discussion in Section 3.4.4).

3.5.2 Object Factory Methods

QconfApi class will provide factory methods for generating PyCL objects, either from UGE data and metadata, or from JSON string (see Figure 19). The factory methods will take UGE version as argument. If this argument is not provided, API object will use qmaster's version as default. The factory methods may be invoked directly by library users, or by other API methods.

```
class QconfApi:
    …
    def generate_queue(self, name=None, data=None, metadata=None, json=None,
       uge_version=None, add_required_data=True):
    …
```

Figure 19: Signature of a factory method for generating PyCL Queue object.

Note that factory methods will be able to generate objects with added default values for required and/or optional keys that are missing from input data.

3.5.3 CRUD and List Methods

Get Methods

API “get” (read) methods will typically take a name as argument and return the corresponding PyCL object. Steps involved in this process will be the following:

1) Take object’s name as argument for the “get” method.
2) Create new instance of the (empty) PyCL wrapper class (with version appropriate for the given qmaster).
3) Invoke appropriate qconf “show object” command. Raise an “ObjectNotFound” exception in case object is unknown to qmaster.
4) Parse output of the qconf command and form Python structure that represents UGE object’s configuration data.
5) Set metadata and data in the PyCL wrapper object and return it to user.

```
all_q = qconf.get_queue('all.q')
print 'Queue JSON Representation: ', all_q.to_json()
```

Figure 20: Example of using API “get” methods, which always return PyCL objects.

Add Methods

The “add” (create) methods will perform appropriate qconf “add” (or “add from file”) command. They will always mimic the original qconf behavior, and will be able to take as input arguments either PyCL objects, or Python structures representing UGE data and metadata, or simply JSON strings. For the dictionary-based UGE objects the “add” process will involve these steps:

1) If needed, create new instance of the PyCL wrapper class from UGE data/metadata, and/or from the given JSON string. PyCL object version will be appropriate for the given qmaster.
2) Verify that all keys are present that must be set by the library user. Raise an “InvalidArgument” PyCL exception if that is not the case.
3) Verify that there are no additional (unknown) keys if optional keys are not allowed. Raise an “InvalidArgument” PyCL exception if that is not the case.
4) Verify that UGE object that is being added does not exist, by invoking qconf “show object” command. Raise an “ObjectAlreadyExists” PyCL exception if that is not the case.
5) Add to the new PyCL object all required keys that are missing, with their default values.
6) Generate UGE object configuration file.
7) Invoke appropriate qconf “add from file” command.
8) Return the new PyCL object to the user.

Note that the above process will allow users to take advantage of UGE default values for most object keys. For example, just like with standard qconf command, one should be able to create new queue by specifying only its name, and taking default values for all other properties.

```
new_q = qconf.add_queue(data={'qname' : 'new.q'})
```

Figure 21: Example of using API “add” method with a dictionary-based UGE object.

In addition to using UGE data and metadata, one should also be able to create new queue by passing its JSON representation, e.g.:

```
new_q = qconf.add_queue(json='{"object_version": "1.0",
"object_class": "ClusterQueue", "data": {"qname": "new.q",
"slots": ["2", "[uge=exec-001=1]"],…}}')
```

Figure 22: Example of using API “add” method with JSON string.

For adding names to UGE configuration (e.g., managers or operators), process will be much simpler, as input arguments are simply Python strings, or list of strings to be added. Such methods will have no return values. However, if any of the input names have already been configured, an “ObjectAlreadyExists” exception will be raised, consistent with other API “add” methods.

Modify Methods

The “modify” (update) methods will perform appropriate qconf “modify from file” command, and will be able to take as input arguments either PyCL objects, or Python structures representing UGE data and metadata, or simply JSON strings. For the dictionary-based UGE objects the “modify” process will involve these steps:

1) If needed, create new instance of the PyCL wrapper class from UGE data/metadata, and/or from the given JSON string. PyCL object version will be appropriate for the given qmaster.
2) Verify that all keys are present that must be set by the library user. Raise an “InvalidArgument” PyCL exception if that is not the case.
3) Verify that there are no additional (unknown) keys if optional keys are not allowed. Raise an “InvalidArgument” PyCL exception if that is not the case.
4) Verify that UGE object that is being updated actually exists, by invoking qconf “show object” command. Raise an “ObjectNotFound” PyCL exception if that is not the case.
5) Add to the new PyCL object all required keys that are missing, with values taken from the old object (i.e., merge new object into the old one).
6) Generate UGE object configuration file.
7) Invoke appropriate qconf “modify from file” command.
8) Return the new (modified) PyCL object to the user.

The above process will allow users to perform updates with partially configured objects. In other words, one should be able to modify objects using only a subset of keys:

```
all_q = qconf.modify_queue(data={'qname' : 'all.q',
       'slots' : ['2', '[uge=exec-001=1]']})
```

Figure 23: Example of using API “modify” method with a dictionary-based UGE object.

Like with the “add” methods, “modify” methods will also take JSON strings, e.g.:

```
all_q = qconf.modify_queue(json='{"object_version": "1.0",
"object_class": "ClusterQueue", "data": {"qname": "all.q",
"slots": ["2", "[uge=exec-001=1]"],…}}')
```

Figure 24: Example of using API “modify” method with JSON string.

Delete Methods

API “delete” methods will take object’s name as argument and invoke qconf “delete” command. Those methods will not have a return value.

```
qconf.delete_queue('new.q')
```

Figure 25: Example of using API “delete” methods.

List Methods

API “list” methods will not take arguments; they will invoke qconf “show list” command appropriate for a given object, and will return a “QconfNameList” object:

```
queue_name_list = qconf.list_queues()
print 'Queue Name List JSON Representation: ', queue_name_list.to_json()
```

Figure 26: Example of using API “list” methods, which always return QconfNameList object.

3.5.4 Support for UGE Upgrades

PyCL API itself will not provide explicit methods for upgrading objects from one version to another, but it will provide support for UGE software upgrades. In other words, library users will be able to write scripts that will perform object and configuration upgrades. In most cases existing objects’ keys will not be removed or renamed, and the following process would work:

1) Take object's old version JSON string representation, and generate corresponding PyCL object (old version).
2) Generate new (upgraded) PyCL object using the old object's UGE data and specifying "add_required_data=True" for the corresponding factory method. This will add keys required for the new PyCL class using with their default values.

Upgrade scripts for more complex cases will have to incorporate appropriate logic that handles those.

3.5.5 Exceptions

In case of any errors PyCL API methods will raise exceptions. All library exceptions will derive from the base QconfException class, which in turn will extend standard Python Exception.

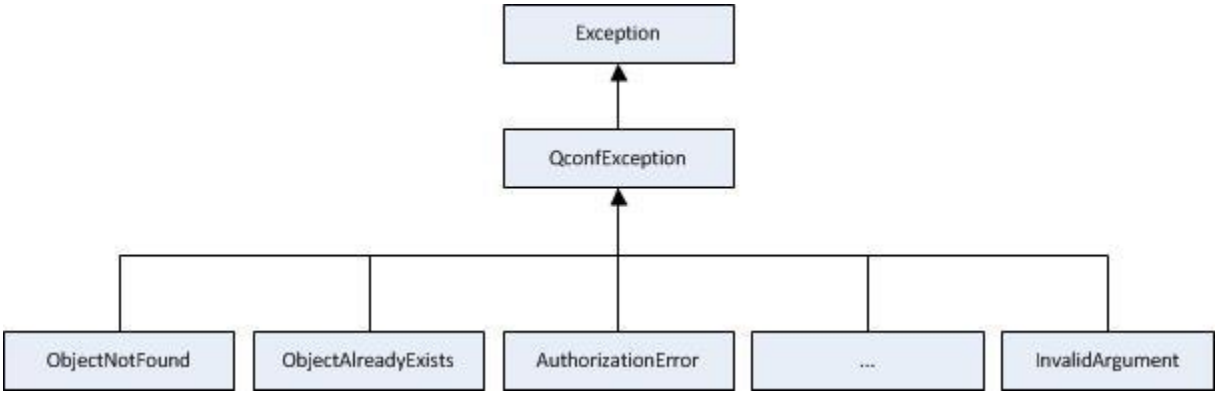

Figure 27: PyCL exceptions will extend QconfException (and standard Python Exception) class.

In those cases where PyCL can determine specific error condition, either by parsing qconf error message or by catching a specific Python error, the library will raise corresponding PyCL exception. For all other cases (either unhandled qconf error, or unexpected Python error) PyCL will raise generic QconfException.

All library exception classes will be initialized with an error message coming either from qconf command error, or from Python exception message. They will also contain unique non-zero error code, which will be used for command line error handling.

3.5.6 Top-Level Module

PyCL API classes, exceptions and object modules will be accessible under the top-level `"uge"` module. Hence, the library users will be able to use Python import statements such as `"from uge import QconfApi"`.

3.5.7 Logging

PyCL logging framework will be based on the standard Python logging modules and will provide the following functionality:

- The library will be able to log messages into log files, as well as send them to the console.
- There will be several different log levels available (e.g., INFO, WARN, DEBUG, ERROR, CRITICAL).
- Logging configuration (not the code implementation) will determine what messages will be logged and where. In other words, variables like logging level will not be hardcoded.
- Logging configuration will be taken from the following sources: configuration file, environment variables (e.g., PYCL_LOG_LEVEL), and command line arguments (for interactive commands).
- Log message format will be configurable, and the system must have ability to prepend timestamp to all messages. For example, log messages might be formatted as follows:

```
<TIMESTAMP> [<LOG LEVEL>] <USER>@<HOST> <LOGGER NAME> (<PID>): <LOG MESSAGE>

12/15/17 11:35:13 [INFO] root@head01 QconfApi (1234): Adding queue new.q to cluster C1
```

3.5.8 Testing

PyCL testing framework will be based on the Python “nose” package [2] and will provide the following functionality:

- Testing will be configurable as far as UGE installation is concerned.
- Testing will be self-reliant (no external setup will be needed other than a valid UGE installation, and running UGE services).
- Testing will be adding, modifying and removing various objects, but will not leave any of its objects in UGE. In other words, the PyCL test suite will restore UGE state after all tests complete.
- Unit tests will include all PyCL objects and API methods.
- Basic successful outcomes for all methods will be tested.
- Testing for various failure scenarios and error handling will be added as problems are found and fixed.

3.5.9 Documentation

All PyCL API and object classes (i.e., those classes intended for users directly) will be documented using Sphinx Python Documentation generator [3]. API documentation will include usage examples for all API methods.

3.6 Command Line Interfaces

Although primary usage of the PyCL library will be through its API, we anticipate that there will also be a need for command line interfaces (CLIs). Those may be used for scripting of various administrative tasks that might not be suitable for using UGE qconf command directly. For example, if managing production clusters requires logging of all configuration changes, PyCL logging capabilities might give an advantage over using the qconf command.

In order to ensure consistency and ease of use, all of the PyCL command line interfaces should have the following functionality:

- All command line tools will be scriptable.
- All CLI classes will be built on top of the PyCL API.
- Commands will return exit status of 0 in those cases where everything worked as expected. Non-zero exit status will be returned in case of any errors or incorrect usage, and it will correspond to error code of the underlying PyCL API exception.
- Clear error message pointing to the cause of problem will be displayed in case of errors. In most cases, this message will be taken directly from the underlying PyCL exception.
- All tools will print help message and exit (exit status of 0) for the following options:

  ```
  -h|--help|-?
  ```

- All tools will print PyCL version information and exit (exit status of 0) for the following options:

  ```
  -v|--version
  ```

- Logging on the screen will be controlled either using environment variable PYCL_LOG_LEVEL, or by providing the following option:

  ```
  --log=<log level>
  ```

- All command parameters may provide “single-dash” flags/options, but they must always offer the “double-dash” syntax (--option=<option value>).

All CLI classes should derive from the base QconfCli class, which will provide set of features and command line options common to all command line tools. Those features include parsing of the command line arguments, error handling, etc.

4 Conclusions

In this document we outlined high-level design for the UGE Python Configuration Library. The design aims to be as detailed as possible in terms of specifying the library functionality and structure of its objects. Nevertheless, we expect that the library implementation phase will likely result in additions or slight modifications of some features.

5 References

[1] For investigating UGE qconf command functionality and features we relied on official UGE Administrators’ guide, as well as on qconf man pages.

[2] Nose package home page: https://pypi.python.org/pypi/nose

[3] Sphinx project home page: http://www.sphinx-doc.org

6 Appendix: PyCL Object Details and API Specification

In this appendix we show PyCL object details: required keys, default UGE values, and corresponding Python types. We also show method signatures for the relevant API methods. Note the following:

- PyCL object keys will be identical to UGE (qconf) keys.
- PyCL objects will use native Python types whenever possible, and will also attempt to convert UGE object values to appropriate types. In those cases where conversion to native types is not possible UGE object values will be interpreted either as strings, or as string lists (UGE object list entries are separated by commas). Standard UGE string keyword “NONE” (or “none”) will be converted to Python None. Similarly, UGE keywords “YES” and “NO” will be converted to Python Boolean values. The library will handle reverse conversions (None to “NONE”, etc.) when creating or modifying UGE objects.
- Some object keys will have to be set by the library user. All other missing keys will be replaced by their default values when creating or modifying UGE objects.
- Default values for certain keys that rely on variables like SGE_ROOT and SGE_CELL may be generated at runtime.
- PyCL will accept any input that results in a valid UGE object configuration.
- When manipulating objects, Python string list is equivalent to a single string where list items are joined by commas. In other words, key value specifications like the ones shown below are equivalent as far as PyCL API is concerned:

```
>>> all_q.data['slots'] = ['1','[uge-exec-001=2]']
>>> all_q.data
{'slots': ['1', '[uge-exec-001=2]']}
>>> all_q.data['slots'] = '1,[uge-exec-001=2]'
>>> all_q.data
{'slots': '1,[uge-exec-001=2]'}
```

  The same is true for corresponding JSON representations.
- Full JSON representation for any PyCL object is based on dictionary that will be formed from metadata dictionary key/value pairs, as well as from UGE object assigned to the reserved key “data” (see Section 3).
- PyCL objects will support JSON representation with both native Python types and with UGE keywords. With native UGE syntax generated JSON strings will have values of “NONE”, “YES” or “NO, rather than null, true or false. Library users will be able to specify JSON representation mode when invoking “to_json()” method on PyCL objects.

QconfApi class constructor signature is shown below:

```
__init__(self, sge_root=None, sge_cell='default', sge_qmaster_port=6444,
       sge_execd_port=6445)
```

6.1 ClusterQueue

UGE data for ClusterQueue objects (“-sq <queue name>”) are represented as Python dictionaries. All of the keys below are required for add/update operations, and are returned by the get operation:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
qname	template	None
hostlist	NONE	None
seq_no	0	0
load_thresholds	np_load_avg=1.75	'np_load_avg=1.75'
suspend_thresholds	NONE	None
nsuspend	1	1
suspend_interval	00:05:00	'00:05:00'
priority	0	0
min_cpu_interval	00:05:00	'00:05:00'
qtype	BATCH INTERACTIVE	'BATCH INTERACTIVE'
ckpt_list	NONE	None
pe_list	make	'make'
jc_list	NO_JC,ANY_JC	['NO_JC','ANY_JC']
rerun	FALSE	False
slots	1	1
tmpdir	/tmp	'/tmp'
shell	/bin/csh	'/bin/csh'
prolog	NONE	None
epilog	NONE	None
shell_start_mode	unix_behavior	'unix_behavior'
starter_method	NONE	None
suspend_method	NONE	None
resume_method	NONE	None
terminate_method	NONE	None
notify	00:00:60	'00:00:60'
owner_list	NONE	None
user_lists	NONE	None
xuser_lists	NONE	None
subordinate_list	NONE	None
complex_values	NONE	None
projects	NONE	None
xprojects	NONE	None
calendar	NONE	None
initial_state	default	'default'
s_rt	INFINITY	float('inf')
h_rt	INFINITY	float('inf')
d_rt	INFINITY	float('inf')
s_cpu	INFINITY	float('inf')
h_cpu	INFINITY	float('inf')
s_fsize	INFINITY	float('inf')
h_fsize	INFINITY	float('inf')
s_data	INFINITY	float('inf')
h_data	INFINITY	float('inf')
s_stack	INFINITY	float('inf')
h_stack	INFINITY	float('inf')
s_core	INFINITY	float('inf')
h_core	INFINITY	float('inf')
s_rss	INFINITY	float('inf')
h_rss	INFINITY	float('inf')
s_vmem	INFINITY	float('inf')
h_vmem	INFINITY	float('inf')

Sample JSON representation for a ClusterQueue object (shortened for simplicity) is shown below:

```
>>> all_q.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ClusterQueue", "modified_on": "2016-04-15T19:30:07.969048",
"data": {"qname": "all.q", "hostlist": "@allhosts", "seq_no": 0, "load_thresholds":
"np_load_avg=1.75",…}}'
```

UGE data for a list of queues (“-sql”) will be returned within the QconfNameList object, which will support all regular Python list features. Sample JSON representation is shown below:

```
>>> queue_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of queues", "data": ["all.q", "long.q",…]}'
```

QconfApi class method signatures relevant to ClusterQueue objects are as follows:

```
ClusterQueue generate_queue(self, name=None, data=None, metadata=None, json=None,
       uge_version=None, add_required_data=True)
ClusterQueue add_queue(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
ClusterQueue modify_queue(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
ClusterQueue get_queue(self, name)
None delete_queue(self, name)
QconfNameList list_queues(self)
```

Note that in order to generate, add, or modify cluster queue, its name can be specified either explicitly, or as part of other input data (queue object, data dictionary or JSON string).

6.2 Cluster Configuration

UGE data for ClusterConfiguration objects (“-sconf [global | <host name>]”) are represented as Python dictionaries. There are no required keys, and arbitrary non-default keys are allowed for add/update operations. However, the default values are different for the “global” versus the host-specific configuration. This will be handled by two sets of defaults for optional keys.

For the “global” configuration default key/value pairs returned by the get operation are shown below:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
execd_spool_dir	$SGE_ROOT/$SGE_CELL/spool	'$SGE_ROOT/$SGE_CELL/spool'
mailer	/bin/mail	'/bin/mail'
xterm	/usr/bin/xterm	'/usr/bin/xterm'
load_sensor	none	None
prolog	none	None

```
epilog                        none                          None
shell_start_mode              unix_behavior                 'unix_behavior'
login_shells                  sh,bash,ksh,csh,tcsh          ['sh','bash','ksh','csh','tcsh']
min_uid                       0                             0
min_gid                       0                             0
user_lists                    none                          None
xuser_lists                   none                          None
projects                      none                          None
xprojects                     none                          None
default_jc                    none                          None
enforce_jc                    false                         False
enforce_project               false                         False
enforce_user                  auto                          'auto'
load_report_time              00:00:40                      '00:00:40'
max_unheard                   00:04:00                      '00:04:00'
reschedule_unknown            00:00:00                      '00:00:00'
loglevel                      log_warning                   'log_warning'
administrator_mail            none                          None
set_token_cmd                 none                          None
pag_cmd                       none                          None
token_extend_time             none                          None
shepherd_cmd                  none                          None
qmaster_params                none                          None
execd_params                  KEEP_ACTIVE=ERROR             'KEEP_ACTIVE=ERROR'
reporting_params              accounting=true reporting=false flush_time=00:00:13 joblog=false
sharelog=00:00:00
                                                            {'accounting' : True, 'reporting' :
False, 'flush_time' : '00:00:13', 'joblog' : False, 'sharelog' : '00:00:00'}

finished_jobs                 0                             0
gid_range                     20000-20100                   '20000-20100'
qlogin_command                builtin                       'builtin'
qlogin_daemon                 builtin                       'builtin'
rlogin_command                builtin                       'builtin'
rlogin_daemon                 builtin                       'builtin'
rsh_command                   builtin                       'builtin'
rsh_daemon                    builtin                       'builtin'
max_aj_instances              2000                          2000
max_aj_tasks                  75000                         75000
max_u_jobs                    0                             0
max_jobs                      0                             0
max_advance_reservations      0                             0
auto_user_oticket             0                             0
auto_user_fshare              0                             0
auto_user_default_project     none                          None
auto_user_delete_time         86400                         86400
delegated_file_staging        false                         False
reprioritize                  0                             0
jsv_url                       none                          None
jsv_allowed_mod               ac,h,i,e,o,j,M,N,p,w          ['ac','h','i','e','o','j','M','N','p',
'w']

cgroups_params                cgroup_path=none cpuset=false mount=false freezer=false
freeze_pe_tasks=false killing=false forced_numa=false h_vmem_limit=false m_mem_free_hard=false
m_mem_free_soft=false min_memory_limit=0
                                                            {'cgroup_path' : None, 'cpuset' :
False, 'mount' : False, 'freezer' : False, 'freeze_pe_tasks' : False, 'killing' : False,
'forced_numa' : False, 'h_vmem_limit' : False, 'm_mem_free_hard' : False, 'm_mem_free_soft' :
False, 'min_memory_limit' : 0}
```

On the other hand, for the host-specific configuration default key/value pairs returned by the get operation are the following:

```
UGE/PYTHON KEY                DEFAULT UGE VALUE             DEFAULT PYTHON VALUE

mailer                        /bin/mail                     '/bin/mail'
xterm                         /usr/bin/xterm                '/usr/bin/xterm'
```

By default ClusterConfiguration keys do not include object's name ("global" or host name). This will be handled by introducing "object_name" metadata key. Sample JSON representation for a ClusterConfiguration object (shortened for simplicity) is shown below:

```
>>> global_conf.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ClusterConfiguration", "object_name": "global", "modified_on":
"2016-04-15T19:30:07.969048", "data": {"execd_spool_dir":
"/opt/tools/uge/default/spool",…}}'
```

UGE data for a list of configurations ("-sconfl") will be returned within the QconfNameList object (see sample JSON representation below):

```
>>> conf_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of configurations", "data": ["global", "uge-exec-001",…]}'
```

Relevant QconfApi class method signatures are as follows:

```
ClusterConfiguration generate_conf(self, name=None,
       data=None, metadata=None, json=None,
       uge_version=None, add_optional_data=False)
ClusterConfiguration add_conf(self, pycl_object=None,
       name=None, data=None, metadata=None, json=None)
ClusterConfiguration modify_conf(self, pycl_object=None,
       name=None, data=None, metadata=None, json=None)
ClusterConfiguration get_conf(self, name=None)
None delete_conf(self, name=None)
QconfNameList list_confs(self)
```

6.3 Scheduler Configuration

UGE data for SchedulerConfiguration objects ("-ssconf") are represented as Python dictionaries. All of the keys below are required for add/update operations, and are returned by the get operation:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
algorithm	default	'default'
schedule_interval	0:0:15	'0:0:15'
maxujobs	0	0
queue_sort_method	load	'load'
job_load_adjustments	np_load_avg=0.50	'np_load_avg=0.50'

```
load_adjustment_decay_time        0:7:30                      '0:7:30'
load_formula                      np_load_avg                 'np_load_avg'
schedd_job_info                   false                       False
flush_submit_sec                  1                           1
flush_finish_sec                  1                           1
params                            none                        None
reprioritize_interval             0:0:0                       '0:0:0'
halftime                          168                         168
usage_weight_list                 wallclock=0.000000,cpu=1.000000,mem=0.000000,io=0.000000
                                                              ['wallclock=0.000000',
'cpu=1.000000', 'mem=0.000000' , 'io=0.000000']

compensation_factor               5.000000                    5.0
weight_user                       0.250000                    0.25
weight_project                    0.250000                    0.25
weight_department                 0.250000                    0.25
weight_job                        0.250000                    0.25
weight_tickets_functional         0                           0
weight_tickets_share              0                           0
share_override_tickets            TRUE                        True
share_functional_shares           TRUE                        True
max_functional_jobs_to_schedule   200                         200
report_pjob_tickets               TRUE                        True
max_pending_tasks_per_job         50                          50
halflife_decay_list               none                        None
policy_hierarchy                  OFS                         'OFS'
weight_ticket                     0.010000                    0.01
weight_waiting_time               0.000000                    0.0
weight_deadline                   3600000.000000              3600000.0
weight_urgency                    0.100000                    0.1
weight_priority                   1.000000                    1.0
fair_urgency_list                 NONE                        None
max_reservation                   0                           0
default_duration                  INFINITY                    float('inf')
backfilling                       ON                          True
prioritize_preemptees             FALSE                       False
preemptees_keep_resources         FALSE                       False
max_preemptees                    0                           0
preemption_distance               00:15:00                    '00:15:00'
preemption_priority_adjustments   none                        None
```

Sample JSON representation for a SchedulerConfiguration object (shortened for simplicity) is shown below:

```
>>> scheduler_conf.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "SchedulerConfiguration", "modified_on": "2016-04-
15T19:30:07.969048", "data": {"algorithm": "default", "schedule_interval":
"0:0:15", "maxujobs": 0, "queue_sort_method": "load",…}}'
```

Relevant QconfApi class method signatures are given as follows (no add/delete/list methods):

```
SchedulerConfiguration generate_sconf(self, data=None,
       metadata=None, json=None, uge_version=None, add_required_data=True)
SchedulerConfiguration modify_sconf(self, pycl_object=None,
       data=None, metadata=None, json=None)
SchedulerConfiguration get_sconf(self)
```

6.4 Execution Host

UGE data for ExecutionHost objects (“-se <host name>”) are represented as Python dictionaries. All of the keys below are required for add/update operations:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
hostname	template	None
load_scaling	NONE	None
complex_values	NONE	None
user_lists	NONE	None
xuser_lists	NONE	None
projects	NONE	None
xprojects	NONE	None
usage_scaling	NONE	None
report_variables	NONE	None
license_constraints	NONE	None
license_oversubscription	NONE	None

Two additional keys (“load_values” and “processors”) are returned by the get operations. Sample JSON representation for an ExecutionHost object (shortened for simplicity) is given below:

```
>>> execution_host.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ExecutionHost", "modified_on": "2016-04-15T19:30:07.969048",
"data": {"hostname": "uge-exec-001", "load_scaling": null, "complex_values":
"m_mem_free=2007.000000M",…}}'
```

UGE data for a list of execution hosts (“-sel”) will be returned within the QconfNameList object:

```
>>> execution_host_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of execution hosts", "data": ["uge-exec-001", "uge-exec-
002",…]}'
```

QconfApi class method signatures relevant to ExecutionHost objects are as follows:

```
ExecutionHost generate_ehost(self, name=None, data=None,
       metadata=None, json=None, uge_version=None, add_required_data=True)
ExecutionHost add_ehost(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
ExecutionHost modify_ehost(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
ExecutionHost get_ehost(self, name)
None delete_ehost(self, name)
QconfNameList list_ehosts(self)
```

In order to generate, add, or modify execution host, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.5 Host Group

UGE data for HostGroup objects (“-shgrp <host group name>”) are represented as Python dictionaries. There are two keys required for add/update operations, and are also returned by the get operations:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
group_name	@template	None
hostlist	NONE	None

Sample JSON representation for a HostGroup object is shown below:

```
>>> host_group.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "HostGroup", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"group_name": "@allhosts", "hostlist": ["uge-exec-001", "uge-exec-002", "uge-exec-
003",…]}}'
```

UGE data for a list of host groups (“-shgrpl”) will be returned within the QconfNameList object:

```
>>> host_group_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of host groups", "data": ["@allhosts",…]}'
```

QconfApi class method signatures relevant to HostGroup objects are as follows:

```
HostGroup generate_hgrp(self, name=None, data=None, metadata=None,
       json=None, uge_version=None, add_required_data=True)
HostGroup add_hgrp(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
HostGroup modify_hgrp(self, pycl_object=None, name=None, data=None,
       metadata=None, json=None)
HostGroup get_hgrp(self, name)
None delete_hgrp(self, name)
QconfNameList list_hgrps(self)
```

In order to generate, add, or modify host group, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.6 Complex Configuration

UGE data for ComplexConfiguration objects (“-sc”) are represented as Python dictionaries, where keys are complex attribute names and values are dictionaries of attribute data. Built-in complex attributes (shown below) are required, but optional attributes are allowed.

```
UGE/PYTHON KEY      DEFAULT UGE COMPLEX DATA

#name               shortcut   type        relop requestable consumable default  urgency aapre
#-------------------------------------------------------------------------------------------
arch                a          RESTRING    ==    YES         NO         NONE     0       NO
calendar            c          RESTRING    ==    YES         NO         NONE     0       NO
cpu                 cpu        DOUBLE      >=    YES         NO         0        0       NO
d_rt                d_rt       TIME        <=    YES         NO         0:0:0    0       NO
display_win_gui     dwg        BOOL        ==    YES         NO         0        0       NO
h_core              h_core     MEMORY      <=    YES         NO         0        0       NO
h_cpu               h_cpu      TIME        <=    YES         NO         0:0:0    0       NO
h_data              h_data     MEMORY      <=    YES         NO         0        0       NO
h_fsize             h_fsize    MEMORY      <=    YES         NO         0        0       NO
h_rss               h_rss      MEMORY      <=    YES         NO         0        0       NO
h_rt                h_rt       TIME        <=    YES         NO         0:0:0    0       NO
h_stack             h_stack    MEMORY      <=    YES         NO         0        0       NO
h_vmem              h_vmem     MEMORY      <=    YES         NO         0        0       NO
hostname            h          HOST        ==    YES         NO         NONE     0       NO
load_avg            la         DOUBLE      >=    NO          NO         0        0       NO
load_long           ll         DOUBLE      >=    NO          NO         0        0       NO
load_medium         lm         DOUBLE      >=    NO          NO         0        0       NO
load_short          ls         DOUBLE      >=    NO          NO         0        0       NO
m_cache_l1          mcache1    MEMORY      <=    YES         NO         0        0       NO
m_cache_l2          mcache2    MEMORY      <=    YES         NO         0        0       NO
m_cache_l3          mcache3    MEMORY      <=    YES         NO         0        0       NO
m_core              core       INT         <=    YES         NO         0        0       NO
m_mem_free          mfree      MEMORY      <=    YES         YES        0        0       YES
m_mem_free_n0       mfree0     MEMORY      <=    YES         YES        0        0       YES
m_mem_free_n1       mfree1     MEMORY      <=    YES         YES        0        0       YES
m_mem_free_n2       mfree2     MEMORY      <=    YES         YES        0        0       YES
m_mem_free_n3       mfree3     MEMORY      <=    YES         YES        0        0       YES
m_mem_total         mtotal     MEMORY      <=    YES         YES        0        0       YES
m_mem_total_n0      mmem0      MEMORY      <=    YES         YES        0        0       YES
m_mem_total_n1      mmem1      MEMORY      <=    YES         YES        0        0       YES
m_mem_total_n2      mmem2      MEMORY      <=    YES         YES        0        0       YES
m_mem_total_n3      mmem3      MEMORY      <=    YES         YES        0        0       YES
m_mem_used          mused      MEMORY      >=    YES         NO         0        0       NO
m_mem_used_n0       mused0     MEMORY      >=    YES         NO         0        0       NO
m_mem_used_n1       mused1     MEMORY      >=    YES         NO         0        0       NO
m_mem_used_n2       mused2     MEMORY      >=    YES         NO         0        0       NO
m_mem_used_n3       mused3     MEMORY      >=    YES         NO         0        0       NO
m_numa_nodes        nodes      INT         <=    YES         NO         0        0       NO
m_socket            socket     INT         <=    YES         NO         0        0       NO
m_thread            thread     INT         <=    YES         NO         0        0       NO
m_topology          topo       RESTRING    ==    YES         NO         NONE     0       NO
m_topology_inuse    utopo      RESTRING    ==    YES         NO         NONE     0       NO
m_topology_numa     unuma      RESTRING    ==    YES         NO         NONE     0       NO
mem_free            mf         MEMORY      <=    YES         NO         0        0       NO
mem_total           mt         MEMORY      <=    YES         NO         0        0       NO
mem_used            mu         MEMORY      >=    YES         NO         0        0       NO
min_cpu_interval    mci        TIME        <=    NO          NO         0:0:0    0       NO
np_load_avg         nla        DOUBLE      >=    NO          NO         0        0       NO
np_load_long        nll        DOUBLE      >=    NO          NO         0        0       NO
np_load_medium      nlm        DOUBLE      >=    NO          NO         0        0       NO
np_load_short       nls        DOUBLE      >=    NO          NO         0        0       NO
num_proc            p          INT         ==    YES         NO         0        0       NO
qname               q          RESTRING    ==    YES         NO         NONE     0       NO
rerun               re         BOOL        ==    NO          NO         0        0       NO
s_core              s_core     MEMORY      <=    YES         NO         0        0       NO
s_cpu               s_cpu      TIME        <=    YES         NO         0:0:0    0       NO
s_data              s_data     MEMORY      <=    YES         NO         0        0       NO
s_fsize             s_fsize    MEMORY      <=    YES         NO         0        0       NO
s_rss               s_rss      MEMORY      <=    YES         NO         0        0       NO
```

```
s_rt                s_rt       TIME        <=    YES         NO         0:0:0    0        NO
s_stack             s_stack    MEMORY      <=    YES         NO         0        0        NO
s_vmem              s_vmem     MEMORY      <=    YES         NO         0        0        NO
seq_no              seq        INT         ==    NO          NO         0        0        NO
slots               s          INT         <=    YES         YES        1        1000     YES
swap_free           sf         MEMORY      <=    YES         NO         0        0        NO
swap_rate           sr         MEMORY      >=    YES         NO         0        0        NO
swap_rsvd           srsv       MEMORY      >=    YES         NO         0        0        NO
swap_total          st         MEMORY      <=    YES         NO         0        0        NO
swap_used           su         MEMORY      >=    YES         NO         0        0        NO
tmpdir              tmp        RESTRING    ==    NO          NO         NONE     0        NO
virtual_free        vf         MEMORY      <=    YES         NO         0        0        NO
virtual_total       vt         MEMORY      <=    YES         NO         0        0        NO
virtual_used        vu         MEMORY      >=    YES         NO         0        0        NO
# >#< starts a comment but comments are not saved across edits --------
```

An example of a Python dictionary representing complex attribute data “slots” from the table above is as follows:

```
>>> slots_attribute = {'shortcut' : 's', 'type' : 'INT', 'relop' : '<=',
'requestable' : True, 'consumable' : True, 'default' : 1, 'urgency' :
1000, 'aapre' : True}
```

Sample JSON representation for a ComplexConfiguration object (shortened for simplicity) is shown below:

```
>>> complex_configuration.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ComplexConfiguration", "modified_on": "2016-04-
15T19:30:07.969048", "data": {"slots": {"shortcut" : "s", "type" : "INT", "relop" :
"<=", "requestable" : true, "consumable" : true, "default" : 1, "urgency" : 1000,
"aapre" : true},…}}'
```

QconfApi class method signatures relevant to ComplexConfiguration objects are as follows:

```
ComplexConfiguration generate_cconf(self, data=None,
      metadata=None, json=None, uge_version=None, add_required_data=True)
ComplexConfiguration modify_cconf(self, pycl_object=None,
      data=None, metadata=None, json=None)
ComplexConfiguration get_cconf(self)
ComplexConfiguration add_cattr(self, name, data)
ComplexConfiguration modify_cattr(self, name, data)
ComplexConfiguration delete_cattr(self, name)
```

6.7 Project

UGE data for Project objects (“-sprj <project name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
name	template	None
oticket	0	0
fshare	0	0
acl	NONE	None
xacl	NONE	None

Sample JSON representation for a Project object is shown below:

```
>>> project.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "Project", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"name": "P1", "oticket": 0, "fshare": 0, "acl": null, "xacl": null}}'
```

UGE data for a list of projects (“-sprjl”) will be returned within the QconfNameList object:

```
>>> project_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of projects", "data": ["P1",…]}'
```

QconfApi class method signatures relevant to Project objects are as follows:

```
Project generate_prj(self, name=None, data=None,
      metadata=None, json=None, uge_version=None, add_required_data=True)
Project add_prj(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
Project modify_prj(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
Project get_prj(self, name)
None delete_prj(self, name)
QconfNameList list_prjs(self)
```

In order to generate, add, or modify project, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.8 User

UGE data for User objects (“-suser <user name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
name	template	None
oticket	0	0
fshare	0	0
delete_time	0	0
default_project	NONE	None

Sample JSON representation for an User object is shown below:

```
>>> user.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "User", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"name": "U1", "oticket": 0, "fshare": 0, "delete_time": 0, "default_project":
null}}'
```

UGE data for a list of users (“-suserl”) will be returned within the QconfNameList object:

```
>>> user_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of users", "data": ["U1",…]}'
```

QconfApi class method signatures relevant to User objects are as follows:

```
User generate_user(self, name=None, data=None,
       metadata=None, json=None, uge_version=None, add_required_data=True)
User add_user(self, pycl_object=None, name=None, data=None, metadata=None,
       json=None)
User modify_user(self, pycl_object=None, name=None, data=None, metadata=None,
       json=None)
User get_user(self, name)
None delete_user(self, name)
QconfNameList list_users(self)
```

In order to generate, add, or modify user, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.9 Access List

UGE data for AccessList objects (“-su <list name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
name	None	None
type	ACL	'ACL'
fshare	0	0
oticket	0	0
entries	NONE	None

Sample JSON representation for an AccessList object is shown below:

```
>>> acl.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "AccessList", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"name": "arusers", "oticket": 0, "fshare": 0, "type": "ACL", "entries": null}}'
```

UGE data for a list of access lists (“-sul”) will be returned within the QconfNameList object:

```
>>> acl_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of access lists", "data": ["arusers",…]}'
```

QconfApi class method signatures relevant to AccessList objects are the following:

```
AccessList generate_acl(self, name=None, data=None,
      metadata=None, json=None, uge_version=None, add_required_data=True)
AccessList add_acl(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
AccessList modify_acl(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
AccessList get_acl(self, name)
[AccessList] add_users_to_acls(self, user_names, access_list_names)
[AccessList] delete_users_from_acls(self, user_names, access_list_names)
None delete_acl(self, name)
QconfNameList list_acls(self)
```

In order to generate, add, or modify access list, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.10 Job Class

UGE data for JobClass objects (“-sjc <job class name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
jcname	template	None
variant_list	NONE	None
owner	NONE	None
user_lists	NONE	None
xuser_lists	NONE	None
A	{+}UNSPECIFIED	'{+}UNSPECIFIED'
a	{+}UNSPECIFIED	'{+}UNSPECIFIED'
ar	{+}UNSPECIFIED	'{+}UNSPECIFIED'
b	{+}UNSPECIFIED	'{+}UNSPECIFIED'
binding	{+}UNSPECIFIED	'{+}UNSPECIFIED'
c_interval	{+}UNSPECIFIED	'{+}UNSPECIFIED'
c_occasion	{+}UNSPECIFIED	'{+}UNSPECIFIED'
CMDNAME	{+}UNSPECIFIED	'{+}UNSPECIFIED'
CMDARG	{+}UNSPECIFIED	'{+}UNSPECIFIED'
ckpt	{+}UNSPECIFIED	'{+}UNSPECIFIED'
ac	{+}UNSPECIFIED	'{+}UNSPECIFIED'
cwd	{+}UNSPECIFIED	'{+}UNSPECIFIED'
dl	{+}UNSPECIFIED	'{+}UNSPECIFIED'
e	{+}UNSPECIFIED	'{+}UNSPECIFIED'
h	{+}UNSPECIFIED	'{+}UNSPECIFIED'
hold_jid	{+}UNSPECIFIED	'{+}UNSPECIFIED'
hold_jid_ad	{+}UNSPECIFIED	'{+}UNSPECIFIED'
i	{+}UNSPECIFIED	'{+}UNSPECIFIED'
j	{+}UNSPECIFIED	'{+}UNSPECIFIED'
js	{+}UNSPECIFIED	'{+}UNSPECIFIED'
l_hard	{+}UNSPECIFIED	'{+}UNSPECIFIED'
l_soft	{+}UNSPECIFIED	'{+}UNSPECIFIED'
masterl	{+}UNSPECIFIED	'{+}UNSPECIFIED'
m	{+}UNSPECIFIED	'{+}UNSPECIFIED'
mbind	{+}UNSPECIFIED	'{+}UNSPECIFIED'
M	{+}UNSPECIFIED	'{+}UNSPECIFIED'
masterq	{+}UNSPECIFIED	'{+}UNSPECIFIED'
N	{+}UNSPECIFIED	'{+}UNSPECIFIED'
notify	{+}UNSPECIFIED	'{+}UNSPECIFIED'
now	{+}UNSPECIFIED	'{+}UNSPECIFIED'
o	{+}UNSPECIFIED	'{+}UNSPECIFIED'
P	{+}UNSPECIFIED	'{+}UNSPECIFIED'
p	{+}UNSPECIFIED	'{+}UNSPECIFIED'
pe_name	{+}UNSPECIFIED	'{+}UNSPECIFIED'
pe_range	{+}UNSPECIFIED	'{+}UNSPECIFIED'
q_hard	{+}UNSPECIFIED	'{+}UNSPECIFIED'
q_soft	{+}UNSPECIFIED	'{+}UNSPECIFIED'
R	{+}UNSPECIFIED	'{+}UNSPECIFIED'
r	{+}UNSPECIFIED	'{+}UNSPECIFIED'
rou	{+}UNSPECIFIED	'{+}UNSPECIFIED'
S	{+}UNSPECIFIED	'{+}UNSPECIFIED'
shell	{+}UNSPECIFIED	'{+}UNSPECIFIED'
t	{+}UNSPECIFIED	'{+}UNSPECIFIED'
tc	{+}UNSPECIFIED	'{+}UNSPECIFIED'
V	{+}UNSPECIFIED	'{+}UNSPECIFIED'
v	{+}UNSPECIFIED	'{+}UNSPECIFIED'

Sample JSON representation for a JobClass object (shortened for simplicity) is shown below:

```
>>> job_class.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "JobClass", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"jcname": "JC1", "variant_list": null,…, "v": "{+}UNSPECIFIED"}}'
```

UGE data for a list of job classes (“-sjcl”) will be returned within the QconfNameList object:

```
>>> job_class_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of job classes", "data": ["JC1",…]}'
```

QconfApi class method signatures relevant to JobClass objects are the following:

```
JobClass generate_jc(self, name=None, data=None,
      metadata=None, json=None, uge_version=None, add_required_data=True)
JobClass add_jc(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
JobClass modify_jc(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
JobClass get_jc(self, name)
None delete_jc(self, name)
QconfNameList list_jcs(self)
```

In order to generate, add, or modify job class, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.11 Resource Quota Set

UGE data for ResourceQuotaSet objects (“-srqs <set name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
name	template	None
description	NONE	None
enabled	FALSE	False
limit	to slots=0	'to slots=0'

Note that the “limit” keyword designates rule definition. Each RQS may have multiple rule definitions, which will be indicated by a Python list.

Sample JSON representation for a ResourceQuotaSet object is shown below:

```
>>> rqs.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ResourceQuotaSet", "modified_on": "2016-04-15T19:30:07.969048",
"data": {"name": "RQS1", "description": null, "enabled": false, "limit": ["users
{user1,user2} hosts {@lx_host} to virtual_free=6g", "users {*} hosts {@lx_host} to
virtual_free=4g"]}}'
```

UGE data for a list of resource quota sets (“-srqsl”) will be returned within the QconfNameList object:

```
>>> rqs_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of resource quota sets", "data": ["RQS1",…]}'
```

QconfApi class method signatures relevant to ResourceQuotaSet objects are the following:

```
ResourceQuotaSet generate_rqs(self, name=None, data=None,
       metadata=None, json=None, uge_version=None, add_required_data=True)
ResourceQuotaSet add_rqs(self, pycl_object=None, name=None,
       data=None, metadata=None, json=None)
ResourceQuotaSet modify_rqs(self, pycl_object=None, name=None,
       data=None, metadata=None, json=None)
ResourceQuotaSet get_rqs(self, name)
None delete_rqs(self, name)
QconfNameList list_rqss(self)
```

In order to generate, add, or modify resource quota set, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.12 Parallel Environment

UGE data for ParallelEnvironment objects (“-sp <name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
pe_name	None	None
slots	0	0
used_slots	0	0
bound_slots	0	0
user_lists	NONE	None
xuser_lists	NONE	None
start_proc_args	NONE	None
stop_proc_args	NONE	None
allocation_rule	$pe_slots	'$pe_slots'
control_slaves	FALSE	None
job_is_first_task	TRUE	True
urgency_slots	min	'min'
accounting_summary	FALSE	False
daemon_forks_slaves	FALSE	False
master_forks_slaves	FALSE	False

Sample JSON representation for a ParallelEnvironment object, shortened for simplicity, is shown below:

```
>>> pe.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ParallelEnvironment", "modified_on": "2016-04-15T19:30:07.969048",
"data": {"pe_name": "PE1", "slots": 100,…, "master_forks_slaves": false}}'
```

UGE data for a list of parallel environments (“-spl”) will be returned within the QconfNameList object:

```
>>> pe_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of parallel environments", "data": ["PE1",…]}'
```

QconfApi class method signatures relevant to ParallelEnvironment objects are the following:

```
ParallelEnvironment generate_pe(self, name=None, data=None,
      metadata=None, json=None, uge_version=None, add_required_data=True)
ParallelEnvironment add_pe(self, pycl_object=None, name=None,
      data=None, metadata=None, json=None)
ParallelEnvironment modify_pe(self, pycl_object=None, name=None,
      data=None, metadata=None, json=None)
ParallelEnvironment get_pe(self, name)
None delete_pe(self, name)
QconfNameList list_pes(self)
```

In order to generate, add, or modify parallel environment, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.13 Share Tree

UGE data for ShareTree object (“-sstree”) is represented as list of Python dictionaries, where dictionaries represent tree nodes. The keys that define tree node completely are “id”, “name”, “type”, “shares”, and “childnodes”. However, tree nodes can be added by specifying only path and number of shares.

Sample JSON representation for a ShareTree object is shown below:

```
>>> share_tree.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "ShareTree", "modified_on": "2016-04-15T19:30:07.969048", "data":
[{"childnodes": [1,2], "type": 0, "id": 0, "shares": 1, "name": "Root"},
{"childnodes": "NONE", "type": 0, "id": 1, "shares": 10, "name": "U1"},
{"childnodes": "NONE", "type": 0, "id": 2, "shares": 25, "name": "U2"}]}'
```

QconfApi class method signatures relevant to ShareTree objects are the following:

```
ShareTree generate_stree(self, data=None, metadata=None, json=None,
       uge_version=None)
ShareTree add_stree(self, pycl_object=None, data=None, metadata=None,
       json=None)
ShareTree modify_stree(self, pycl_object=None, data=None, metadata=None,
       json=None)
ShareTree modify_or_add_stree(self, pycl_object=None, data=None, metadata=None,
       json=None)
ShareTree get_stree(self)
ShareTree get_stree_if_exists(self)
None delete_stree(self)
None delete_stree_if_exists(self)
ShareTree add_stnode(self, path, shares)
ShareTree delete_stnode(self, path)
```

In the above list methods modify_or_add_stree(), get_stree_if_exists(), and delete_stree_if_exists() do not throw exception if share tree does not exist.

6.14 Calendar

UGE data for Calendar objects (“-scal <calendar name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
calendar_name	None	None
year	NONE	None
week	NONE	None

Sample JSON representation for a Calendar object is shown below:

```
>>> calendar.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "Calendar", "modified_on": "2016-04-15T19:30:07.969048", "data":
{"calendar_name": "CAL1", "year": null, "week": "1-2"}}'
```

UGE data for a list of calendars (“-scall”) will be returned within the QconfNameList object:

```
>>> calendar_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of calendars", "data": ["CAL1",…]}'
```

QconfApi class method signatures relevant to Calendar objects are the following:

```
Calendar generate_cal(self, name=None, data=None, metadata=None, json=None,
      uge_version=None, add_required_data=True)
Calendar add_cal(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
Calendar modify_cal(self, pycl_object=None, name=None, data=None,
      metadata=None, json=None)
Calendar get_cal(self, name)
None delete_cal(self, name)
QconfNameList list_cals(self)
```

In order to generate, add, or modify calendar, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.15 Checkpointing Environment

UGE data for CheckpointingEnvironment objects (“-sckpt <name>”) are represented as Python dictionaries. Optional keys are not allowed. The keys required for add/update operations, and also returned by the get operations are the following:

UGE/PYTHON KEY	DEFAULT UGE VALUE	DEFAULT PYTHON VALUE
ckpt_name	None	None
interface	userdefined	'userdefined'
ckpt_command	none	None
migr_command	none	None
restart_command	none	None
clean_command	none	None
ckpt_dir	/tmp	'/tmp'
signal	none	None
when	sx	'sx'

Sample JSON representation for a CheckpointingEnvironment object is shown below:

```
>>> ckpt.to_json()
'{"object_version": "1.0", "modified_by": "sveseli", "uge_cluster": "C1",
"object_class": "CheckpointingEnvironment", "modified_on": "2016-04-
15T19:30:07.969048", "data": {"ckpt_name": "CKPT1", "interface": "userdefined",…,
"when": "sx"}}'
```

UGE data for a list of checkpointing environments (“-sckptl”) will be returned within the QconfNameList object:

```
>>> ckpt_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of checkpointing environments", "data": ["CKPT1",…]}'
```

QconfApi class method signatures relevant to CheckpointingEnvironment objects are the following:

```
CheckpointingEnvironment generate_ckpt(self, name=None,
       data=None, metadata=None, json=None, uge_version=None,
       add_required_data=True)
CheckpointingEnvironment add_ckpt(self, pycl_object=None,
       name=None, data=None, metadata=None, json=None)
CheckpointingEnvironment modify_ckpt(self, pycl_object=None,
       name=None, data=None, metadata=None, json=None)
CheckpointingEnvironment get_ckpt(self, name)
None delete_ckpt(self, name)
QconfNameList list_ckpts(self)
```

In order to generate, add, or modify checkpointing environment, its name can be specified either explicitly, or as part of other input data (data dictionary or JSON string).

6.16 Managers

List of UGE managers (“-sm”) will be represented using QconfNameList object. Sample JSON representation is shown below:

```
>>> manager_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of managers", "data": ["ugeadmin", "sveseli"]}'
```

Relevant QconfApi class method signatures are the following:

```
QconfNameList list_managers(self)
QconfNameList add_managers(self, manager_names)
QconfNameList delete_managers(self, manager_names)
```

Methods for adding and deleting managers will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

6.17 Operators

List of UGE operators (“-so”) will be represented using QconfNameList object. Sample JSON representation is shown below:

```
>>> operator_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of operators", "data": ["ugeadmin", "sveseli"]}'
```

Relevant QconfApi class method signatures are the following:

```
QconfNameList list_operators(self)
QconfNameList add_operators(self, operator_names)
QconfNameList delete_operators(self, operator_names)
```

Methods for adding and deleting operators will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

6.18 Submit Hosts

List of UGE submit hosts (“-ss”) will be represented using QconfNameList object. Sample JSON representation is shown below:

```
>>> submit_host_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of submit hosts", "data": ["shost-001", "shost-002"]}'
```

Relevant QconfApi class method signatures are the following:

```
QconfNameList list_shosts(self)
QconfNameList add_shosts(self, host_names)
QconfNameList delete_shosts(self, host_names)
```

Methods for adding and deleting submit hosts will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.

6.19 Admin Hosts

List of UGE admin hosts (“-sh”) will be represented using QconfNameList object. Sample JSON representation is shown below:

```
>>> admin_host_list.to_json()
'{"object_version": "1.0", "uge_cluster": "C1", "object_class": "QconfNameList",
"description": "list of admin hosts", "data": ["ahost-001", "ahost-002"]}'
```

Relevant QconfApi class method signatures are the following:

```
QconfNameList list_ahosts(self)
QconfNameList add_ahosts(self, host_names)
QconfNameList delete_ahosts(self, host_names)
```

Methods for adding and deleting admin hosts will return list of names after requested operation has been completed. In other words, the “add” method will return list that contains old names together with names that have been added, while the “delete” method will return list of old names without names that have been deleted.