Managing Scientific Workflows in Python with `pyutilib.workflow`

William E. Hart*

September 5, 2011

Abstract

We describe the capabilities of the `pyutilib.workflow` software package. This package provides Python classes that provide an intuitive interface for defining and executing scientific workflows. Further, `pyutilib.workflow` is a native Python package, so it can be used to define workflows within Python software applications. Additionally, `pyutilib.workflow` includes a utility for creating a command-line driver that execute workflows as subcommands of a command-line script.

*Sandia National Laboratories, Data Analysis and Informatics Department, PO Box 5800, Albuquerque, NM 87185; `wehart@sandia.gov`

1 Introduction

Scientific workflow is an increasingly popular strategy for managing complex scientific computation processes. Workflows allow scientists to automate data transformations, describe complex computational procedures, and parallelize these analyses. Scientific workflow systems are closely related to workflow models used in business process management systems. The key difference is that scientific workflows focus on the transformation of data through algorithms, whereas business workflows focus on scheduling and execution of tasks.

Many of the workflow packages developed in Python are best described as business workflow systems. For example, packages like `django-workflows` and `zope.app.workflow` provide workflows for web content management. The following native Python workflow packages appear to be suitable for scientific workflows:

- *Pyphant*: This is a framework for scientific data analysis. A computational analysis is defined by a graph of processing steps, which is managed with a workflow engine.
- *Python Workflow Engine*: This is a simple workflow engine that was initially based on the workflow engine used in the ACE project.
- *Spiff Workflow*: This package is designed around the workflow patterns defined at `http://www.workflowpatterns.com`.
- *Ruffus*: This is a lightweight python module to run computational pipelines (See `http://www.ruffus.org.uk/`).
- *PaPy*: A lightweight python package that manages parallel computational pipelines (see `http://muralab.org/PaPy/`)

Other packages like VisTrails [4] and Weaver [1] also support the management of scientific workflows in Python, though they rely on external software packages to execute these workflows.

This report describes the `pyutilib.workflow` (PW) package, which supports the definition and execution of scientific workflows. The following key features of PW that distinguish it from other Python workflow tools:

- PW is a self-contained package that was designed to be used within other software applications. Although PW depends on several other PyUtilib Python packages, it does not rely on external software packages to execute PW workflows.
- PW defines a workflow through the interaction of task objects, rather than an explicit definition of a workflow graph. For example, a connection between two tasks is created by setting an output in one task equal to an input in the other.
- a PW workflow (or task) can be treated as a functor that executes with the given arguments and returns a dictionary of computed data.

- PW tasks can be created as plugin components that can be dynamically created with a task factory. This supports the modular definition of tasks and workflows, and it allows the definition of workflows to be isolated from the definition of task classes.
- PW workflows can be initialized with command-line arguments. Further, PW includes a command-line driver that can executed named workflows using a subcommand syntax that is commonly used in command-line tools (e.g. `svn`).

The remainder of this manuscript provides a detailed description of the capabilities in PW. We include many examples that illustrate how PW objects interact to define and execute workflows, and we discuss the command-line driver that can execute workflows with values specified by command-line arguments.

2 Managing Workflows

2.1 Overview

Figure 1 provides a graphical illustration of the components of a workflow. A *workflow* is comprised of one or more computational steps, which we call a *task* or *component*. A task maps a set of input data into a set of output data. Input and output data are managed with *port* objects, and tasks are linked together with *connectors* that define a connection from an output port in one task to an input port for another. These connections form a directed acyclic graph (DAG), which defines how task executions need to be coordinated to correctly execute the entire workflow.

2.2 A Simple Example

The main goal of PW is to support the definition of workflows in an intuitive manner using Python objects. There are two fundamental classes defined by PW that are used to define a workflow: `Task` and `Workflow`. A user defines tasks by creating subclasses of the `Task` class. For example, the following task computes the sum of its two inputs:

```
class TaskA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y
```

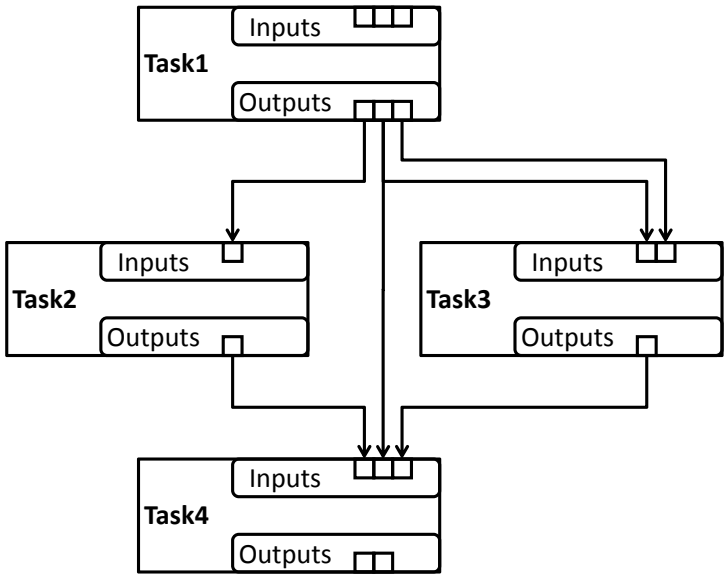

Figure 1: A graphical illustration of a workflow with four tasks. Black lines between tasks represent connectors, and square boxes in the tasks represent the input and output ports.

The `Task` class defines the `inputs` and `outputs` attributes that are used to respectively declare input and output ports. These declarations must be included in the task constructor, since the inputs and outputs are treated as static task properties by PW.

The task computation is performed by the `execute` method, which must be defined by the user. Note that the input and output values are attributes of the task object. This simplifies the syntax for users developing task computations by allowing them to treat task data as they would in any other Python object. PW initializes the value of these attributes before calling `execute`, and it interrogates the task afterwards to set the value of the output ports.

The following Python code creates the `TaskA` object, creates a `Workflow` object, initializes the workflow with this task, and then executes the workflow with input values:

```
A = TaskA()
w = pyutilib.workflow.Workflow()
w.add(A)
```

```
print w(x=1, y=3)
```

Note that the workflow defines a functor, which is executed with keyword arguments that are mapped to the task inputs. This functor returns an `Options` object, which is a glorified Python dict class. The output of printing the workflow results is:

```
Options:
    z = 4
```

2.3 Defining Connections

The previous example was a trivial illustration of the setup and execution of a workflow. In practice, workflows will be defined by constructing two or more tasks that are linked together. Suppose we wish to compute the expression:

$$z = 2 * x + y.$$

We can employ `TaskA` to perform the sum, and the following task to double the value of x:

```
class TaskB(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.Z = 2*self.X
```

The following Python code creates the `TaskA` and `TaskB` objects, links the output of B to the input of A, and then creates and executes a workflow:

```
A = TaskA()
B = TaskB()
A.inputs.x = B.outputs.Z

w = pyutilib.workflow.Workflow()
w.add(A)
print w(X=1, y=3)
```

The connection between `TaskA` and `TaskB` is defined with the command

```
A.inputs.x = B.outputs.Z
```

The syntax transparently creates a `Connector` object that connects the `Z` output of `TaskB` to the `x` input of `TaskA`. This greatly simplifies the declaration of connections when compared with other Python workflow packages. Note that this mechanism allows an output port to be connected to one or more input ports. The default setup of ports allows an input port to only connect to a single output port. (See Section 2.4 for further discussion.)

As in our earlier example, the workflow is created by constructing a `Workflow` object and then adding tasks to it. Note, however, that in this example only `TaskA` was added. The `Workflow` object traverses the connections between tasks to identify all tasks connected to the task that is added. Consequently, only a single task in a workflow needs to be added to the `Workflow` object.

Note that the functor defined by the workflow has a slightly different API in this example; it uses inputs `X` and `y`. To understand why, consider Figure 2, which shows the workflow in this example. Tasks `TaskA` and `TaskB` are connected to each other, but also to a start and end task. The start and end tasks are constructed when a `Workflow` object loads the workflow. The start task contains outputs that correspond to every input port that is not connected to an output port. Similarly, the end task contains inputs that correspond to every output port that is not connect to an input port. In this way, the inputs and outputs of the workflow are automatically defined.

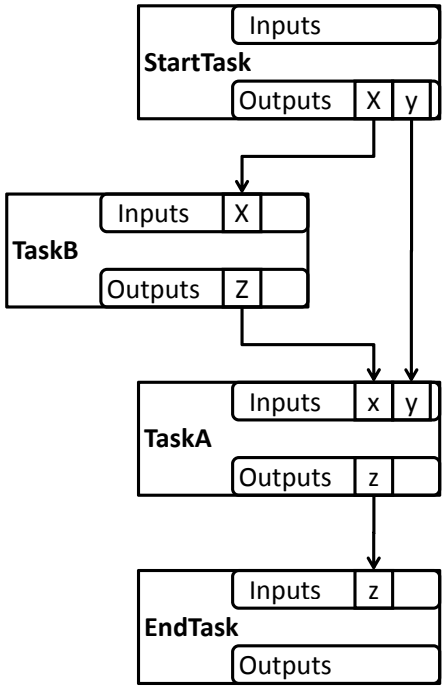

Figure 2: An illustration of the workflow defined with tasks `TaskA` and `TaskB`.

To see further implications of this logic, suppose that `TaskC` is used instead of `TaskB`:

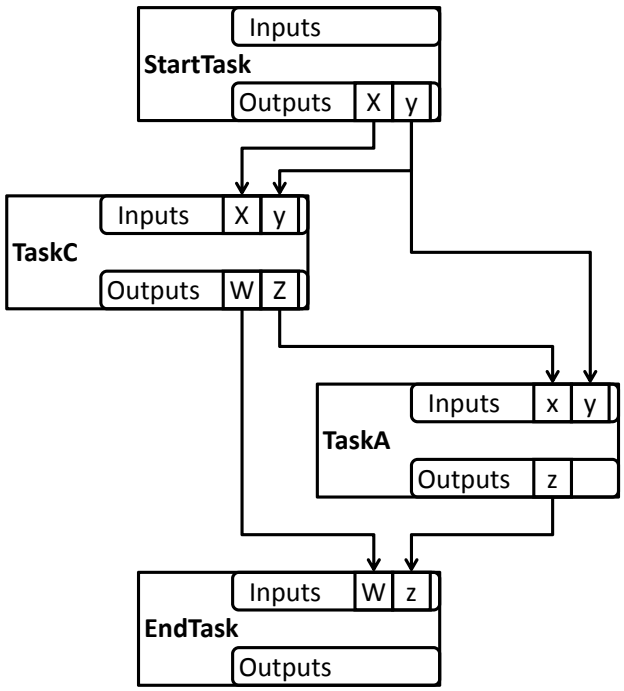

Figure 3: An illustration of the workflow defined with tasks `TaskA` and `TaskC`.

```
class TaskC(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.inputs.declare('y')
        self.outputs.declare('W')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.W = self.X+self.y
        self.Z = 2*self.W
```

Figure 3 shows the workflow for this example. The setup and execution of this task does not change. However, the input `y` is now used by both tasks `TaskA` and `TaskC`. Further, the

output W is now included in the final results. The output of printing the workflow results is:

```
Options:
    W = 4
    z = 11
```

Similarly, the following example uses TaskD instead of TaskB:

```
class TaskD(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.inputs.declare('y')
        self.inputs.declare('a', constant=True)
        self.outputs.declare('W')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.W = self.X+self.y+self.a
        self.Z = 2*self.W
```

The input a is a constant value that is not included in the outputs of the start task. However, this value can be set directly using the TaskD object. The output of printing the workflow results is:

```
Options:
    W = 104
    z = 211
```

2.4 Input Ports with Multiple Connections

The action constructor option for the Port class defines how input connections are used to compute the input value. The default action is store, which indicates that the connector value is stored in the port. This behavior reflects the previous examples, and it is well-suited for workflows where there is a direct correspondence between output ports and input ports.

However, contexts often arise in practice where a suite of tasks needs to be computed and their results are analyzed together. For example, consider TaskD which generalizes TaskA to sum an arbitrary number of inputs:

```
class TaskD(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x', action='append')
        self.outputs.declare('z')

    def execute(self):
```

```
        """Compute the sum of the inputs."""
        self.z = sum(self.x)
```

Note that the input port `x` is defined with the `append` action, which configures it to create a list of input values.

The following example use `TaskD` to define a workflow with inputs from `TaskE`, which generates a random integer value:

```
class TaskE(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('Y')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.Z = random.randint(1,self.Y)

random.seed(123809870128)
D = TaskD()
for i in range(100):
    E = TaskE()
    D.inputs.x = E.outputs.Z

w = pyutilib.workflow.Workflow()
w.add(D)
print w(Y=100)
```

In this example, `TaskE` objects are created and connected to the `TaskD` object with the command:

```
D.inputs.x = E.outputs.Z
```

The input `x` port is configured to append inputs to a list, and no special syntax is needed to indicate how the connections are configured between the `x` port and the `Z` ports.

The `map` action can also be specified to define an input as a dictionary with keys that are the task ids from the connection that generated the values. For example, this can be used to associate data generated in different branches of a workflow. The following example uses this associate to define a dictionary, which is the final result:

```
class TaskF1(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('a',)
        self.inputs.declare('aval')
        self.outputs.declare('a', self.inputs.a)
        self.outputs.declare('aval', self.inputs.aval)
```

```
    def execute(self):
        pass

class TaskF2(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('A',)
        self.inputs.declare('Aval')
        self.outputs.declare('A', self.inputs.A)
        self.outputs.declare('Aval', self.inputs.Aval)

    def execute(self):
        pass

class TaskG(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x', action='map')
        self.inputs.declare('y', action='map')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = {}
        for key in self.x:
            self.z[ self.x[key] ] = self.y[key]

F1 = TaskF1()
F2 = TaskF2()
G = TaskG()
G.inputs.x = F1.outputs.a
G.inputs.y = F1.outputs.aval
G.inputs.x = F2.outputs.A
G.inputs.y = F2.outputs.Aval

w = pyutilib.workflow.Workflow()
w.add(G)
print w(a='a', aval=1, A='A', Aval=2)
```

Tasks `TaskF1` and `TaskF2` simply map their inputs to outputs. Their outputs are connected to two inputs in `TaskG`, and these inputs are used to create a dictionary. The output of this computation is:

```
Options:
    z = {'a': 1, 'A': 2}
```

Normally, an input port with the `store`, `append` or `map` action cannot be evaluated if any of the output ports connected to it is not in the `ready` state. However, the `store_any`,

append_any and map_any actions allow any or all of the inputs to be in a non-ready state. When the store_any action is specified, the value is simply taken from the first connection that is in the ready state. When the map_any action is specified, then a dictionary is formed from all connections that are in the ready state. Similarly, the append_any action appends all values from connections inthe ready state.

2.5 Using Workflows as Tasks

A key feature of PW is the ability to use workflows as components of other workflows. This is possible because Workflow is a subclass of Task.

For example, consider the following workflows that are defined with TaskA and TaskC:

```
A = TaskA()
C = TaskC()
A.inputs.x = C.outputs.Z

w1 = pyutilib.workflow.Workflow()
w1.add(A)

AA = TaskA()
AA.inputs.x = w1.outputs.W
AA.inputs.y = w1.outputs.z

w2 = pyutilib.workflow.Workflow()
w2.add(AA)

print w2(X=1, y=3)
```

Workflow w1 is the workflow defined in the previous example. This object is used to define workflow w2, which uses TaskA to sum the outputs of w1: W and z. The output of executing w2 is

```
Options:
    z = 15
```

2.6 Initializing Port Values

Task ports are initialized through the execution of a workflow, and through the explicit specification of port values. The simplest way to specify port values is to define them directly. For example, consider the following variation of the example in Section 2.2:

```
A = TaskA()
w = pyutilib.workflow.Workflow()
w.add(A)
A.inputs.x = 1
A.inputs.y = 3
print w()
```

The workflow is constructed as before, but the values of ports `x` and `y` are defined explicitly, and the execution of the workflow does not specify these values.

PW also supports the initialization of port values with command-line options. The goal of this capability is to facilitate the use of PyUtilib in command-line applications, by allowing command-line options to be used to directly initialize a workflow. The following example is a simple extension of the example in Section 2.2.

```
class TaskAA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.add_argument('--x', dest='x', type=int)
        self.add_argument('--y', dest='y', type=int)
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y

AA = TaskAA()
w = pyutilib.workflow.Workflow()
w.add(AA)
w.set_options(['--x=1', '--y=3', '--bad=4'])
print w()
```

Some additional logic is added to the `TaskAA` class to specify the command-line options. In this example, the `set_argument` method is used to initialize a workflow with a list of option strings. This syntax mimics the format of data provided by `sys.argv`. Again, the execution of the workflow does not specify these values.

Note that port values specified in these ways are viewed as default values for the port. When a port value is computed from input connections, then the port value will be overriden if the input connections provide a non-trivial value. For example, if the port action is `store`, then the value will be overriden if the input connection has a value other than `None`. Similarly, if the port action is `append` or `map`, then the value will be overriden if one or more of the input connections are not None.

Additionally, port values are redefined by the workflow keyword options. For example, in the following script we initialize input ports for `TaskAA`, which are then redefined when the workflow is executed:

```
AA = TaskAA()
w = pyutilib.workflow.Workflow()
w.add(AA)
w.set_options(['--x=1', '--y=3'])
print w(y=4)
```

The output of this script is

```
Options:
   z = 5
```

which reflects the fact that the value of `y` was redefined by the workflow keyword option.

2.7 The Task Factory

PW leverages the PyUtilib Component Architecture [3] to support the definition of a task factory. The PW task factory allows users to create plugin tasks on the fly without requiring knowledge of where these tasks are defined. This capability exposes a variety of standard tasks that are defined in PW , and it can be used to create tasks that are defined by third-party libraries in a standard manner.

The `TaskFactory` object defined in PW is a functor. This functor can be used to create a task that has been registered as a plugin. For example, the `Selection_Task` class is registered with the string 'workflow.selection', and it can be instantiated as follows:

```
task = pyutilib.workflow.TaskFactory('workflow.selection')
```

Section 5 describes the predefined tasks that are provided with PW.

A plugin task is created as a subclass of the `TaskPlugin` class. This registers this task as a plugin with the PyUtilib Component Architecture. The only additional step required for a plugin task is to use the `alias` declaration to define the string that is used to create this task in the task factory.

For example, the following code defines the task `PluginTaskA` that is registered with the string 'TaskA':

```
class PluginTaskA(pyutilib.workflow.TaskPlugin):

    pyutilib.component.core.alias('TaskA')

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y
```

Note that the only difference with the definition of `TaskA` is the specification of the base class and the `alias` declaration.

The following Python code creates the `PluginTaskA` object, creates a `Workflow` object, initializes the workflow with this task, and then executes the workflow with input values:

```
A = pyutilib.workflow.TaskFactory('TaskA')
w = pyutilib.workflow.Workflow()
w.add(A)
print w(x=1, y=3)
```

This has the same logical steps as the example in Section 2.2. The only difference is that the task is created by the task factory.

3 Control Flow Tasks

The basic functionality provided by PW can be characterized as a data flow. Each task represents a transformation of data in input ports to data in output ports. These tasks are networked together in a data flow graph, in which tasks form a directed acyclic graph where data flows from the start task(s) to the final task(s).

PW extends this functionality by providing control flow logic. Tasks include special ports, input and output control ports that can be used to limit the execution of tasks. An output control port is connected to one or more input control ports. If an output control port is set to the *ready* state, then the tasks connected to this with an input control port can be executed. Otherwise, these tasks are blocked until the output control port changes state.

For example, the `Selection_Task` class is a predefined task whose inputs are a dictionary, `data`, and an indexing value, `index`. This task returns `selection`, which is simply the value `data[index]`. This task can be used to switch the execution based on the indexed value. For example:

```
class TaskA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y

B = pyutilib.workflow.TaskFactory('workflow.selection')
A = TaskA()
A.inputs.x = B.outputs.selection
w = pyutilib.workflow.Workflow()
w.add(B)

print w(index='a', y=100, data={'a':1, 'b':2})
w.reset()
print w(index='b', y=100, data={'a':1, 'b':2})
```

This generates the following output:

```
Options:
    z = 101
Options:
```

```
    z = 102
```

The `Switch_Task` class is a predefined task that provides a similar functionality in this example. However, rather than switching the data value, this class switches the control flow for downstream tasks. For example:

```
class TaskA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x', constant=True)
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y

class TaskZ(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('z', action='store_any')
        self.outputs.declare('z', self.inputs.z)

    def execute(self):
        pass

B = pyutilib.workflow.TaskFactory('workflow.switch')
A1 = TaskA()
A1.inputs.x = 1
B.add_branch('a', A1)
A2 = TaskA()
A2.inputs.x = -2
B.add_branch('b', A2)
Z = TaskZ()
Z.inputs.z = A1.outputs.z
Z.inputs.z = A2.outputs.z
w = pyutilib.workflow.Workflow()
w.add(B)

print "Branch a"
print w(value='a', y=100)
w.reset()
print "Branch b"
print w(value='b', y=100)
```

This generates the following output:

```
Branch a
Options:
    z = 101
Branch b
```

```
Options:
    z = 98
```

The `Branch_Task` class provides a simpler version of the same process executed by the `Switch_Task` class. This class switches the control flow for two downstream tasks. For example:

```
Options:
    x = -1
Options:
```

Here, the branches for `TaskA` and `TaskB` are specified with a branch value that is a boolean.

4 Defining Task Resources

There are many contexts in which task execution is dependent on the availability of external resources. For example, data files may need to be available, a database may need to be unlocked, or a software license may need to be free. PW allows these constraints on workflow execution to be represented with `Resource` objects that represent the state of a dependent resource. A resource may or may not be available, and the workflow can lock and unlock a resource as it employs it for execution.

PW defines the `ExecutableResource`, which allows a user to specify an executable that is automatically found by searching the `PATH` environment. If the specified executable is not found, then it is unavailable for execution in a workflow. This resource also includes a utility method for applying this executable with command-line arguments.

The following example illustrates the use of this resource to define a task that lists all of the files in a specified directory:

```
class TaskH(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('dir')
        self.outputs.declare('list')
        self.add_resource(pyutilib.workflow.ExecutableResource('ls'))

    def execute(self):
        self.resource('ls').run(self.dir, logfile=currdir+'logfile')
        self.list = []
        for line in open(currdir+'logfile','r'):
            self.list.append( line.strip() )
        self.list.sort()

H = TaskH()
w = pyutilib.workflow.Workflow()
w.add(H)
print w(dir=currdir+'dummy')
```

A key role of resource objects is that they can limit the execution of tasks. The `availability` method in a resource object is queried to see if a resource can be allocated. The following example illustrates this functionality with a simple `BusyResource` class that is busy the first time it is queried:

```
from pyutilib.workflow import *

class BusyResource(Resource):

    def __init__(self, name=None):
        resource.Resource.__init__(self)
        self._counter = 1

    def available(self):
        if self._counter > 0:
            print "BUSY",self._counter
            self._counter -= 1
            return False
        return True

class TaskA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.outputs.declare('x', self.inputs.x)

    def execute(self):
        pass

A = TaskA()
A.add_resource(BusyResource())
w = pyutilib.workflow.Workflow()
w.add(A)
```

The first time that task `A` is queried, this resource is not available. Note that the PW workflow execution process currently does not allow tasks to block indefinitely. If all tasks have blocked, then the workflow execution will immediately terminate.

5 Predefined Tasks

The following sections describe the task plugins that are defined by PW, and we provide an example of how a task plugin is defined.

5.1 Selection Task

The `workflow.selection` task has the following inputs:

- `data`: a dictionary

- `index`: an index key in the dictionary

This task returns the value of the dictionary with the specified index key.

Note that this task does not fail gracefully if the index key is not defined in the dictionary. An exception will occur that will terminate the execution of the workflow.

6 The Task Driver

The PW task driver provides a facility for creating a command-line utility that can execute PW plugin tasks. The task driver is inspired by command-line tools like `svn` that allow users to specify subcommands that have independent command-line arguments. The PW task driver can be easily configured to execute different tasks and workflows as subcommands within a command-line application.

Consider the following two task classes:

```
class PluginTaskZ(pyutilib.workflow.TaskPlugin):

    pyutilib.component.core.alias('TaskZ')

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.add_argument('--x', dest='x', type=int)
        self.add_argument('--y', dest='y', type=int)
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y

class PluginTaskY(pyutilib.workflow.TaskPlugin):

    pyutilib.component.core.alias('TaskY')

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.inputs.declare('Y')
        self.add_argument('--X', dest='X', type=int)
        self.add_argument('--Y', dest='Y', type=int)
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.Z = self.X * self.Y
```

Note that these are plugin tasks that can be created with the `TaskFactory` functor. The PW task driver can only execute tasks and workflows that are defined as plugins.

Suppose that the `TaskZ` and `TasY` are defined in the file `task_yz.py`. The following script creates a task driver, activates two these two tasks and illustrates the results of parsing two sets of command-line arguments:

```
import tasks_yz

driver = pyutilib.workflow.TaskDriver()
driver.register_task('TaskZ')
driver.register_task('TaskY')

print driver.parse_args(['TaskZ','--x=3','--y=4'])
print driver.parse_args(['TaskY','--X=3','--Y=4'])
```

However, the true value of the task driver is in the definition of a command-line utility. For example, the following script defines a command-line utility that can execute tasks `TaskZ` and `TaskY`:

```
import pyutilib.workflow
import tasks_yz

driver = pyutilib.workflow.TaskDriver()
driver.register_task('TaskZ')
driver.register_task('TaskY')

print driver.parse_args()
```

This script creates the task driver and then parses the `sys.argv` command-line arguments. Suppose that this script is in the file `driver1.py`. Then the following command-line illustrates the execution of task `TaskZ`:

```
python driver1.py TaskZ --x=3 --y=4
```

which generates the following output:

```
Options:
   z = 7
```

The task driver constructor includes several options for declaring the script name and associated documentation that will be printed when the `--help` option is specified:

- **prog** - The name of the script.
- **description** - A short description of the script's functionality.
- **epilog** - Additional documentation that is printed after the command-line options are described.

The following script uses these options to illustrate the help information that is printed by the task driver:

```
import pyutilib.workflow
```

```
import tasks_yz

driver = pyutilib.workflow.TaskDriver(prog='myprog',
   description='This is the description of this task driver',
   epilog="""**********************
This is more text
that describes this command driver. Note

that the format of the epilog string is preserved in the
help
output!
**********************
""")
driver.register_task('TaskZ')
driver.register_task('TaskY')

print driver.parse_args()
```

Suppose that this script is in the file `driver2.py`. Then the following command-line illustrates the execution with the `--help` option:

```
python driver2.py --help
```

which generates the following output:

```
usage: myprog [-h] {TaskY,TaskZ} ...

This is the description of this task driver

positional arguments:
  {TaskY,TaskZ} Sub-commands
    TaskZ
    TaskY

optional arguments:
  -h, --help show this help message and exit

**********************
This is more text
that describes this command driver. Note

that the format of the epilog string is preserved in the
help
output!
**********************
```

Furthermore, the `--help` option can be used to print information about a specific subcommand. The command

```
python driver2.py TaskZ --help
```

generates the following output:

```
usage: myprog TaskZ [-h] [--x X] [--y Y]
```

```
optional arguments:
  -h, --help show this help message and exit
  --x X
  --y Y
```

7 Discussion

A major driver for the development of the PW is the TEVA-SPOT Toolkit [2], which supports research on sensor placement optimization for water security applications. TEVA-SPOT uses the PW task driver to define the `sptk` script, which can execute a variety of different workflows that represent different strategies for sensor placement optimization.

The fact that PW provides a self-contain facility for defining and executing workflows is particularly important for TEVA-SPOT. This code is targeted for distribution on desktop computers, and PW provides a convenient mechanism for flexibly developing new sensor placement strategies that can be executed without a cumbersome workflow management system. Parallel execution of PW workflows is a natural extension of the current capability, which would not require a signficant extension of the current class definitions and workflow syntax.

Finally, here are some notes concerning the current status of development in PW:

- PW includes a variety of methods managing parsers used to initialize tasks. These methods were intended to simplify the setup of commands using workflows. However, these methods have not proven terribly useful in practice. Consequently, we could imagine deprecating this feature of PW unless clear use cases arise.

- The PW execution logic is simply a method of the `Workflow` class. It would be worth exploring how this could be generalized to (a) support threaded parallelism and (b) interface with third-party grid- or cloud- computing workflow engines. This would provide a nice extensibility of this capability while preserving the simple Pythonic interface that PW provides.

- A preliminary resource class for files has been developed, but simple use-cases for this class have not been flushed out.

- Control flow tasks for looping and other more advanced capabilities are not currently provided, but these will probably be developed as the need arises.

Acknowledgements

We are grateful to John Siirola for discussing the design of multi-task connectors, which strongly influenced the design currently employed in `pyutilib.workflow`. Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned

subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] P. Bui, L. Yu, and D. Thain, *Weaver: Integrating distributed computing abstractions into scientific workflows using python*, in Challenges of Large Applications in Distributed Environments at ACM HPDC 2010, June 2010.

[2] W. E. Hart, J. Berry, R. Murray, C. A. Phillips, L. A. Riesen, and J.-P. Watson, *SPOT: A sensor placement optimization toolkit for drinking water contaminant warning system design*, Tech. Rep. SAND2007-4393, Sandia National Laboratories, 2007.

[3] W. E. Hart and J. D. Siirola, *The pyutilib component architecture*, Tech. Rep. SAND2010-2516, Sandia National Laboratories, May 2010.

[4] *Vistrails*. `http://www.vistrails.org`, 2010.