The PyUtilib Component Architecture

William E. Hart* John Siirola†

April 26, 2010

Abstract

We describe the PyUtilib Component Architecture (PCA). The PCA is derived from the Trac component architecture, and it supports advanced features like non-singleton components, namespaces and caching of component interactions. The PCA includes an independent, self-contained framework core that can be easily integrated into other applications, as well as a variety of extension packages with commonly used components.

*Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; wehart@sandia.gov

†Sandia National Laboratories, Exploratory Simulation Technologies Department, PO Box 5800, Albuquerque, NM 87185; jdsiiro@sandia.gov

Contents

1 Introduction

Component Based Software Engineering (CBSE) has become one of the leading approaches to developing complex extensible software systems [5]. CBSE implementations frequently rely on concepts from both object oriented programming (OOP) and event-driven programming. Unlike traditional OOP where classes and polymorphism are used to manage related data-driven objects, CBSE leverages classes and polymorphism to represent related functional interfaces and programmatic *services*. The central idea underlying CBSE is *equivalence of service*; that is, the separation of the declaration of component interfaces from their implementation. This allows for more flexible software design that encourages modularity of component interface and definitions. Furthermore, this segregation allows for explicit management of the interactions among components. We can begin to imagine software components as commodities that can be integrated into applications in a much more flexible and dynamic manner.

A variety of mature, general purpose environments exist for defining and managing components (e.g., the CORBA Component Model [4] and the Common Component Architecture [1]). Although Python interfaces have been developed for some of these environments, a variety of native Python component environments have also been developed, including Zope [13], Envisage [3], Trac [11], yapsy [12] and SprinklesPy [9].

This report describes the PyUtilib Component Architecture (PCA). The PCA is derived from the Trac component framework [11], and it is included in the PyUtilib software package [8]. Our development of the PCA was motivated by our experience with a variety of scientific computing applications, which led to the following requirements for PCA:

- *Independent, self-contained framework core*: Many component architectures are embedded in larger software frameworks (e.g. Zope, Trac), which make it difficult to extract and use just the software packages related to the component architecture.

- *Non-Singleton components*: The computational science applications that motivate PCA require both singleton components (which have a single unique instance) and non-singleton components (which have many unique instances).

- *Namespaces*: Using components in large software projects requires management across multiple libraries. Namespaces are needed to effectively manage components in these complex software projects.

- *Caching*: Components need to support applications where component interfaces are called thousands or millions of times. Thus, caching of this interaction is needed to minimize the overhead of the component architecture.

- *Loading from EGGs*: Support for loading EGG files is invaluable in dynamic applications. Further, loading components from EGG files provides another level of modularity to the management of software applications.

A key guiding principle behind our development of the PCA is a focus on simplicity and flexibility. Our goal is to minimize the burden placed on application developers for both adding the PCA to a project and maintaining PCA-based applications. One consequence is that the PCA explicitly does not provide some advanced capabilities, like interface validation and interface adaption, that are available in more heavyweight component architectures such as Zope or Envisage.

The remainder of this manuscript is divided into the following sections. Section 2 provides a tutorial introduction to the use of PCA classes, and Section 3 provides a detailed description of PCA capabilities. Section 4 describes PyUtilib extension packages that support specific components based on the PCA. This section motivates these packages and provides examples of their use. Section 5 discusses how the PCA has fundamentally influenced the design of the Coopr software project.

2 The PyUtilib Component Architecture

The PyUtilib Component Architecture (PCA) is included in the PyUtilib software package [8]. It is based on the Trac plugin framework [11] and was initially motivated by our need for an independent, self-contained plugin framework for scientific computing applications. The core of the PCA is provided through a small set of classes within PyUtilib's `pyutilib.component.core` package. This section provides a tutorial introduction of the PCA as well as a detailed description of the PCA classes and their functionality.

2.1 PCA Definitions

There are different notions of software components [6], so we begin by providing some definitions. The relationships among these terms is illustrated in Figure 1. A *plugin* is a class that implements a set of related methods in the context of an application. Thus, a plugin can be described as a component definition. A *service* is an instance of a plugin class, so services are component instances. A service can be an instance of either a singleton or non-singleton plugin. There is exactly one service for a singleton plugin (and that service is instantiated automatically), whereas there can be multiple services of non-singleton plugins.

An *interface* class defines a type that a plugin uses to register its capabilities. A plugin class includes declarations that denote that it implements one-or-more interfaces. An interface is defined by the methods and data that are used. However, the PCA does not enforce this interface or support interface conversions (see Zope [13] and Envisage [3] for examples of plugin frameworks that support this functionality).

A software application or a component can declare *extension points* that other components can *plug in* to. An extension point is defined with respect to a specific *interface* class. Thus, a service that supports an interface plugs into an extension point for that interface. In this way, extension points provide a generic mechanism for applications to employ the functionality provided by other services.

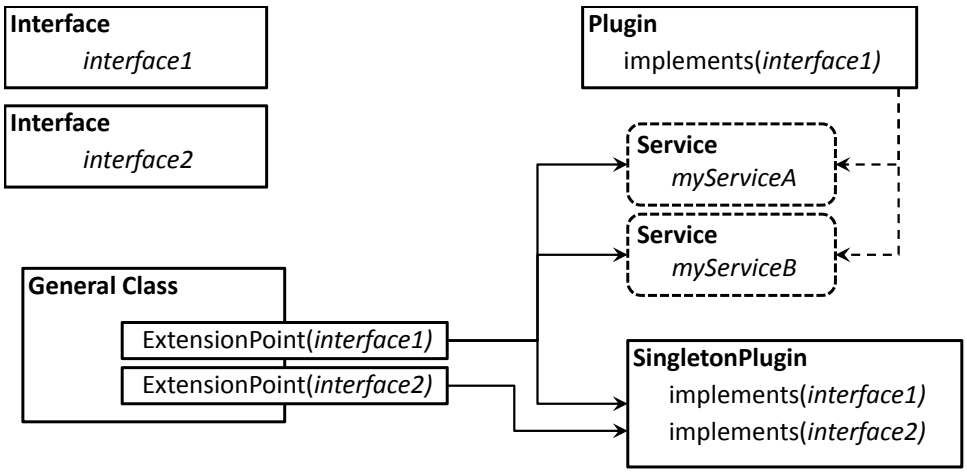

Figure 1: An illustration of how classes within the PCA relate to one another. Interface classes are independent declarations of APIs. Plugin classes can declare that they implement an Interface's API, and extension points declare that they require a specific Interface API. Both singleton and non-singleton plugins can be used, but services for non-singleton plugins are explicitly constructed.

This mechanisms supports a flexible, modular programming paradigm that enables software applications to be extended in a dynamic manner. The PCA includes a global component registry and a framework for automating the execution of plugin services. All plugins and interfaces automatically register themselves with the registry. This registry then acts as a broker, dynamically providing extension points with the appropriate matching services at run time. Thus, an application developer can define extension points without knowing how they will be implemented, and plugin developers can register extensions without needing to know the details of how – or where – they are employed. This capability facilitates the dynamic registration and application of components within large software systems.

2.1.1 Relationship to the Trac Component System

The general design of PCA is adapted from Trac [11]. The PCA generalizes the Trac component architecture by supporting namespace management of plugins, non-singleton plugins and caching of interface interactions. For those familiar with Trac, the following classes roughly correspond with each other:

Trac Class Name	PyUtilib Class Name
Interface	Interface
ExtensionPoint	ExtensionPoint
Component	SingletonPlugin
ComponentManager	PluginEnvironment

The `PluginEnvironment` class is used to manage plugins, but unlike Trac this class does not need to be explicitly constructed.

2.2 A Simple Example

Figure 2 provides a simple example that is adapted from the description of the Trac component architecture [10]. This example illustrates the three main steps to setting up a plugin:

1. Defining an interface
2. Declaring extension points
3. Defining classes that implement the interface.

This example begins by defining `ITaskObserver`, a subclass of `Interface`. Although it is not required to define methods in an interface, these declarations provide documentation for plugin developers. We then declare a `TaskList` that manages a dictionary of tasks and descriptions. The `TaskList` creates an `ITaskObserver` extension point, and when a task is added to the task list, it calls all services that implement the `ITaskObserver` interface. Finally, we define a `TaskPrinter` as a singleton plugin. The `TaskPrinter` implements the `ITaskObserver` interface, and when called prints the task name and description. As the `TaskPrinter` is a singleton plugin, the PCA automatically instantiates and registers a single `TaskPrinter` service.

Assuming the module in Figure 2 is saved as `task.py`, then the following Python script illustrates how this plugin is used:

```
from task import *

# Construct a TaskList object and then add several tasks.
task_list = TaskList()
task_list.add('Make coffee', 'Need to make some coffee')
task_list.add('Bug triage', 'Double-check all issues')
```

This script generates the following output:

```
Task: Make coffee
      Need to make some coffee
Task: Bug triage
      Double-check all issues
```

3 PCA Plugin Classes

The PCA consists of the following core classes:

pyutilib.component.core.Interface Subclasses of this class declare plugin interfaces that are registered in the PCA.

```
# A simple example that manages a task list.  An observer
# interface adds actions that occur when a task is added.
from pyutilib.component.core import *

# An interface class that defines the API for plugins that
# observe when a task is added.
class ITaskObserver(Interface):

    def task_added(name, description):
        """Called when a task is added."""

# The task list application, which declares an extension point
# for observers.  Observers are notified when a new task
# is added to the task list.
class TaskList(object):
    observers = ExtensionPoint(ITaskObserver)

    def __init__(self):
        """The TaskList constructor, which initializes the list"""
        self.tasks = {}

    def add(self, name, description):
        """Add a task, and notify the observers"""
        self.tasks[name] = description
        for observer in self.observers:
            observer.task_added(name, description)

# A plugin that prints information about tasks when they
# are added.
class TaskPrinter(SingletonPlugin):
    implements(ITaskObserver)

    def task_added(self, name, description):
        print 'Task:', name
        print '     ', description
```

Figure 2: A simple example of PCA components.

pyutilib.component.core.ExtensionPoint A class used to declare extension points, which can access services with a particular interface.

pyutilib.component.core.Plugin Subclasses of this class declare plugins, which can be used to implement interfaces within the PCA.

pyutilib.component.core.SingletonPlugin Subclasses of this class declare singleton plugins, for which a single service is constructed.

pyutilib.component.core.PluginEnvironment A class that maintains the registries for interfaces, extension points, plugins and services.

pyutilib.component.core.PluginGlobals A class that maintains global data concerning the set of environments that are currently being used.

pyutilib.component.core.PluginError The exception class that is raised when errors arise in the PCA.

The following sections provide a detailed description of how these classes are used in the PCA.

3.1 Interfaces and Extension Points

A subclass of the `Interface` class is used to declare extension points in an application. The `ExtensionPoint` class is used to declare an extension point and to retrieve information about the plugins that implement the specified interface. For example, the following is a minimal declaration of an interface and extension point:

```
class IMyInterface(Interface):
    """An interface subclass"""

ep = ExtensionPoint(IMyInterface)
```

Note that the `IMyInterface` class is not required to define the API of the interface. The PCA does not enforce checking of API conformance for plugins, and hence any declaration in the `IMyInterface` class would be ignored. Additionally, note that an instance of `IMyInterface` is not created; the `IMyInterface` class is simply used to declare a type that is used to index related plugins.

An instance of `ExtensionPoint` can be used to iterate through all extensions, or to search for an extension that matches a particular keyword. For example, the following code iterates through all extensions and applies the `pprint` method:

```
for service in ep:
  service.pprint()
```

If you wish to know the number of services that are registered with an extension point, you can call the standard `len` function:

```
print len(ep)
```

Several other methods can be used to more carefully select services from an extension point. The `extensions` method returns a Python `set` object that contains the services:

```
#
# This loop iterates over all services, just the same
# as when an the iterator method is used (see above).
#
for service in ep.extensions():
  service.pprint()
```

The Python `__call__` method provides a convenient shorthand for this same function. Thus, the following is equivalent:

```
for service in ep():
  service.pprint()
```

These methods have two optional arguments that control the selection of services. The `all` keyword indicates whether the set returned by `extensions` includes all disabled services.

```
#
# This loop iterates over all services, including
# the 'disabled' services.
#
for service in ep.extensions(all=True):
  service.pprint()
```

It is convenient to explicitly support enabling and disabling services in many applications, though services are enabled by default. Disabled services remain in the registry, but by default they are not included in the set returned by an extension point.

The PCA can also support *named services*, which requires that the services have a `name` attribute. Service names are not required to be unique. For example, when multiple instances of a non-singleton plugin are created, then these services can be accessed as follows:

```
#
# A simple plugin that implements the IMyInterface interface
#
class MyPlugin(Plugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="myname"

#
# Another simple plugin that implements the IMyInterface interface
#
class MyOtherPlugin(Plugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="myothername"

#
# Constructing services
#
service1 = MyPlugin()
service2 = MyPlugin()
service3 = MyOtherPlugin()

#
# A function that iterates over all IMyInterface services, and
# returns the MyPlugin instances (which are service1 and service2).
#
def get_services():
    ep = ExtensionPoint(IMyInterface)
    return ep("myname")
```

In some applications, there is a one-to-one correspondence between service names and their instances. In this context, a simpler syntax is to use the `service` method:

```
class MySingletonPlugin(SingletonPlugin):
  implements(IMyInterface)

  def __init__(self):
      self.name="mysingletonname"

ep = ExtensionPoint(IMyInterface)
ep.service("mysingletonname").pprint()
```

The `service` method raises a `PluginError` if there is more than one service with a given name. Note, however, that this method returns `None` if no service has been registered with the specified name.

Note that an integer cannot be used to select a service from an extension point. Services are not registered in an indexable array, so this option is disallowed.

3.2 Plugins

PCA plugins are subclasses of either the `Plugin` or `SingletonPlugin` classes. Subclasses of `Plugin` need to be explicitly constructed, but otherwise they do not need to be registered; simply constructing a subclass of `Plugin` invokes the registration of that instance. Similarly, simply declaring a subclass of `SingletonPlugin` invokes both the construction and registration of this component.

PCA plugins are registered with different interfaces using the `implements` function, which is a static method of `Plugin`. Note that a plugin can be registered with more than one interface. Further, a service can be applied to different extension points independently, but it can maintain state information that impacts its use across different extension points.

The default behavior of the PCA is to ignore the declarations in an interface class, but the `implements` function includes an `inherit` keyword can be used to define a plugin that inherits interface methods. For example:

```
class IMyInterface(Interface):
    def print(self):
        print "This is the default print method"

    def add(self, x):
        return x+2

class MyPlugin(Plugin):
    implements(IMyInterface, inherit=True)

    def add(self, x):
        return x+3
```

In this example, the `MyPlugin` class implements the `IMyInterface` interface. Since the `inherit` keyword is `True`, the `MyPlugin` class inherits the `print` method. Thus, `MyPlugin` has a complete implementation of the `IMyInterface` interface.

Although this behavior is generally useful, the API for PCA intentionally does not make interface inheritance the default behavior. When inheritance is used, a developer can get into trouble if they mistype the name of a plugin method. When this occurs, the interface method is used, without any notification to the user. This could easily lead to erroneous plugin behavior that is quite difficult to track down.

When `Plugin` classes are instantiated or `SingletonPlugin` classes are declared, the resulting class service is registered in global PCA data (see below). The `Plugin` class includes several methods for controlling this registration. The `disable` and `enable` methods provide a simple mechanism for controlling whether a service is returned with associated extension points. These methods clear the PCA caches that are associated with these extension points

to ensure that the extension points are correctly setup. The `activate` and `deactivate` methods respectively add and remove the service from the global environment. This has a similar effect as `enable` and `disable`, except that after deactivation the service is no longer associated with the PCA global data, while after `disable` the service is still registered but flagged as not active.

3.3 Environments

The `PluginEnvironment` class defines namespaces that contain component services and interfaces. These namespaces provide a mechanism for organizing component services in an extensible manner. Applications can define new namespaces that contain their services without worrying about conflicts with services defined in other Python libraries.

A global registry of environments is maintained by the `PluginGlobals` class. This registry is a stack of environments, and the top of this stack defines the current environment. When an interface is declared, its namespace is the name of the current environment. For example:

```
#
# Declare an interface in the current environment
#
class Interface1(Interface):
    pass

#
# Set the current environment to 'new_environ'
#
PluginGlobals.push_env( "new_environ" )

#
# Declare an interface in the 'new_environ' environment
#

class Interface2(Interface):
    pass

#
# Go back to the original environment
#
PluginGlobals.pop_env()
```

Component services are automatically registered in namespaces in two ways. First, for each interface that the service implements, the service is registered in the namespace in which the interface was declared. Second, a service is registered in the namespace in which its corresponding plugin class is declared.

For example, consider the code in Figure 3. When `Plugin1` is instantiated, this service is registered in the following environments:

```
#
# Declare Interface1 in namespace env1
#
PluginGlobals.push_env("env1")

class Interface1(Interface):
    pass

#
# Declare Interface2 in namespace env2
#
PluginGlobals.push_env("env2")

class Interface2(Interface):
    pass

PluginGlobals.pop_env()

#
# Declare Plugin1 in namespace env3
#
PluginGlobals.push_env("env3")

class Plugin1(Plugin):

    implements(Interface1)
    implements(Interface2)
    implements(Interface1,"env4")

PluginGlobals.pop_env()
```

Figure 3: Illustration of how plugin declarations are related to component environments.

```
env1 for Interface1
env2 for Interface2
env4 for Interface1
env3
```

The last registration is the default, since a service is always registered in the environment where its plugin class is declared. Note that `env4` namespace is declared explicitly in this example.

3.4 Global Component Data

Global component data in PCA is managed in the `PluginGlobals` class. This class contains a variety of static methods that are used to access this data:

default_env This method returns the default environment, which is constructed when the PCA is loaded.

env This method returns the current environment if no argument is specified. Otherwise, it returns the specified environment.

push_env,pop_env These methods respectively push a new environment onto the environment stack and pop the current environment from the stack.

services This method returns the component services in the current environment (or the named environment if one is specified).

singleton_services This method returns the singleton component services in the current environment (or the named environment if one is specified).

load_services Load services using `IPluginLoader` extension points (see Section 4.1).

pprint This method provides a text summary of the registered interfaces, plugins and services.

clear This method empties the environment stack and defines a new default environment. This setup then bootstraps the configuration of the `pyutilib.component.core` environment. Note that this does not clear the component registry; in practice that may not make sense since it is not easy to reload modules in Python.

4 PCA Extensions

In addition to the core component framework, PCA includes implementations for a variety of components that support commonly used functionality. These extensions of PCA are available in PyUtilib packages separate from the PCA core. This emphasizes the modularity of the PCA, and it illustrates how to define PCA components that are automatically registered as part of an application. The following sections briefly describe these PCA extensions.

4.1 Component Loaders

PCA components can be loaded from either Python modules or Python eggs. This capability supports the runtime extension of the PCA, which has proven very powerful in frameworks like Trac. Component services for loading are defined in the `pyutilib.component.loader` package. The core PCA defines extension points that use these services, which can be applied as follows:

```
import sys
import os
env = sys.environ["PATH"]
PluginGlobals.load_services(path=env.split(os.sep))
```

In this example, the user's `PATH` environment is used to define the list of directories that are searched for Python modules and eggs.

The `load_services` takes two other optional arguments that control how components are loaded. The `name_re` argument can be used to define a regular expression that filters the files in the directories that are searched. The following shows how to specify that services starting with *my* are loaded:

```
PluginGlobals.load_services(name_re="my.*")
```

By default, when services are loaded they are disabled. This facilitates the management of services in complex applications using configuration files (see below). The `auto_disable` flag can be used to automatically activate services:

```
PluginGlobals.load_services(auto_disable=False)
```

4.2 Registering Executables

The `pyutilib.component.executable` package defines the `ExternalExecutable` plugin, which is used to define services that provide information about external executables. Services declare the executable name and user documentation, and then service methods indicate whether the executable is enabled (i.e. whether it is found, and the path of the executable:

```
service = ExternalExecutable(name='ls',
            doc='A utility to list file in Unix operating systems')

service.enabled()
# Returns True if the executable is found on the user path.

service.get_path()
# Returns a string that defines the path to this executable,
# or None if service is disabled.
```

The registration process is simplified with the `pyutilib.services` package, which includes the `register_executable` function:

```
import pyutilib.services

pyutilib.services.register_executable('zip')
```

This function searches the user's `PATH` environment for the `zip` executable (or `zip.exe` on Windows machines).

A developer can use the `registered_executable` function to the access the absolute path of a registered executable. If the executable is not found in the user's `PATH`, then this returns `None`. Also, if no executable is specified, then this function returns a list of all registered executables.

4.3 Temporary Files

The `pyutilib.component.config` packages provides a services for managing temporary files. The `TempfileManager` object is a component service whose methods can be used to create and cleanup temporary files; for convenience, this object is accessible from the `pyutilib.services` package.

The main method in this service is `create_tempfile`, which can create a temporary file with a specified suffix and prefix in a specified directory:

```
import pyutilib.services

pyutilib.services.create_tempfile(prefix='myfile',
                                  suffix='.txt', dir='/home/jdoe')
```

By default, this service creates unique filenames. However, if the `sequential_files` method is called, then the body of the temporary files will be an integer that is incremented every time a temporary file is created. Although these filenames may not be unique, this sequential naming scheme may make it easier to diagnose errors in a complex application.

This service keeps track of the temporary files that it creates. This allows an application developer to avoid this bookkeeping, and instead rely on this service to delete temporary files with the `clear_tempfiles` method. Furthermore, a developer can explicitly declare a file as temporary using the `add_tempfile` method, thereby allowing this service to delete it.

4.4 Options and Configuration Files

The `pyutilib.component.config` package defines interfaces and plugins for managing service options. The `Configuration` service is used to manage the global configuration of all services. This class coordinates with `Option` services. Plugins can declare options with the `declare_option` method, which registers these options with the `Configuration` service. This service reads and writes options to configuration files (using Python's `ConfigParser` package).

This package also declares the `ManagedPlugin` and `ManagedSingletonPlugin` classes, which are plugin base classes that include options that can be used to enable or disable services using the `Configuration` service.

4.4.1 Configuration Files

A PCA configuration file consists of a list of sections. Each section is lead by a `[section]` header, and a section contains a list of `name = value` entries. For example, the following configuration file consists of two sections with four option values:

```
# COMMENT
[globals]
a = 1
b = /dev/null
c = 1,2,3
[a.b]
zz = 4.5
```

PCA plugin classes declare component options with the `declare_option` method, which is defined in `pyutilib.component.doc`. These options can be initialized with a configuration file. For example, the following plugin declares four options.

```
class PluginWithOptions(Plugin):
    def __init__(self):
        declare_option("a")
        declare_option("b")
        declare_option("c")
        declare_option("zz",section='a.b')
```

The default configuration section for an option is `[globals]`, but the option declaration can specification the section name. For example, the configuration file described above can be used to initialize the `PluginWithOptions` plugin.

The `Configuration` class manages loading and storing configuration data for the options that are registered by PCA services. This class defines the following methods:

clear Clear the configuration data.

load Load configuration data from a file.

pprint Print a simple summary of the configuration data.

save Write configuration data to a file.

sections Return a list of the sections that have been loaded.

summarize Summarize the options that have been registered with the PCA.

Once data is loaded with the `load` method, the `sections` method can be used to provide a list of the sections that were loaded. The Python `__contains__` method can also be used to check if a section was loaded:

```
from pyutilib.component.config import *

config = Configuration()

# Load configuration data
config.load('config.ini')

# Check if the 'globals' section was loaded
if 'globals' in config:
    print "The 'globals' section was loaded"

# Get the 'globals' section
section = config['globals']
```

The Python `__getitem__` method is used at the end of this example to get the data for the 'globals' section. Section data consists of a dictionary that maps option names to value strings. Note that the `Configuration` class automatically loads this data into the corresponding options that have been registered with the PCA.

4.4.2 Declaring Options

The `declare_option` creates an `Option` object that is a data member in a plugin. The standard syntax for this function is to specify the option name, which is used to define an attribute in the plugin with the same name:

```
class TmpPlugin(SingletonPlugin):

  declare_option('x')
```

The `local_name` keyword can be used to specify a different name for this attribute within the plugin. For example, consider:

```
class TmpPlugin(SingletonPlugin):

  declare_option('x', local_name='y')
```

In this example, the option is declared with name `x` in the PCA registry, but it has attribute name `y` within this plugin. The `default` keyword defines the default value of an option, and the `doc` option specifies a document string that describes the option; this information is used by the `Configuration` class when printing option summary information.

As noted earlier, the `section` keyword can be used to specify the section in configuration data that this option is expected. The `section_re` keyword supports a more generic mechanism. If the `section_re` is specified with a regular expression, then this option will be initialized from any section that matches this regular expression. If sections match and contain data for this option, then the last section specified in the configuration data will be used to initialize this option.

Finally, the `cls` keyword specifies the option type. Option types are described in the next section.

4.4.3 Options Types

The default option type is `Option`. These options treat option values as strings, even when they could be interpreted as numeric values. The `BoolOption`, `IntOption` and `FloatOption` types respectively interpret option values as booleans, integers and floating point values. The `OptionError` exception is raised if the option value is not the appropriate value.

The `FileOption` type interprets the option value into a path. A relative path is converted to an absolute path using the path for the configuration file. Thus, a user can load file path data from any directory but specify the file data relative to the path of the application configuration. In addition, this option type also supports a `directory` keyword that can be used to specify how relative paths are resolved. The `ExecutableOption` type is an extension of `FileOption` that confirms that the file can be executed. If the file name does not include path information, then PCA will search for the executable using the user's `PATH` environment before initializing this option.

The `DictOption` type supports an interface to all options in a section. For example, consider:

```
class TmpPlugin(Plugin):

    options = DictOption(section="bar")
```

The `options` object will be populated by all data in the `bar` section. For example, if the names `a` and `b` are defined in this section, then they can be referenced as `options.a` and `options.b`. Similarly, data can be inserted into the `bar` section by simply specifying the value of attributes of this object:

```
options.c = 2
```

4.4.4 Using Options in Services

It is convenient for singleton plugins to declare options as part of the class definition:

```
class Plugin1(SingletonPlugin):

  declare_option("x")
```

This type of declaration makes sense since there is a single instance of the class `Plugin1`. For non-singleton plugins, this type of declaration would make the *same* option data available to all instances of the plugin. To declare different options for different non-singleton plugin instances, it suffices to execute `declare_option` within the plugin constructor:

```
class Plugin2(Plugin):

    def __init__(self, sec):
        declare_option("x", section=sec)
```

Note that this example allows the `Plugin2` services to distinguish the configuration of these different `x` options by specifying different sections in the configuration file. Although this is not required, this is often desirable in practice.

4.4.5 Managed Services

The PCA supports explicit management of services using the configuration management technology described in this section. The `ManagedPlugin` and `ManagedSingletonPlugin` classes include an `enable` option that controls whether their corresponding services are activated. When services are loaded from EGG files and modules, they are disabled by default. The `Services` section of a configuration file can be used to activate these plugins:

```
[Services]
plugin1 = True
plugin3 = True
```

This allows an application administrator to install a variety number of application services that a user selectively enables.

4.5 Other Extensions

The following extension packages include plugins and applications interfaces for PCA applications. These extension packages are less mature, and consequently they are not documented in detail right now.

pyutilib.component.config This extension package defines a plugin that manages logging of PCA actions.

pyutilib.component.app This package defines a simple application class that can be used as the basis of a component-based application. This application class provides support for managing configuration from a configuration file, and for managing logging activity.

5 Discussion

A major driver for the development of the PCA is the Coopr optimization software [2]. Coopr defines a variety of component interfaces that can be used to customized and extend Coopr's management of optimization solvers. This includes interfaces for components that

- write files that define an optimization problem
- convert files into a format that is compatible with an optimizer
- execute an optimizer
- read files that define optimizer results and execution status

The PCA has had a fundamental impact on the design of Coopr because it supports a new software design for optimization frameworks.

The typical object oriented approach for optimization software is to use classes and class inheritance. For example, the OPT++ [7] optimization software library defines base classes with different characteristics (e.g. differentiability), and a concrete optimization solver is instantiated as a subclass of an appropriate base class. In this context, the base class can be viewed as defining the interface for the solvers that inherit from it.

Coopr components leverage the PCA to separate the declaration of component interfaces from their implementation. For example, the interface to optimization solvers are again declared with a class. However, solver plugins are not required to be subclasses of the interface class. Instead, they are simply required to provide the same interface methods. Consequently, Coopr can be extended and configured in a modular manner that is qualitatively different from other optimization frameworks. The PCA allows Coopr to dynamically construct optimization strategies and combine independently-developed modeling, reformulation, preprocessing, and optimization approaches in a manner that is substantially more flexible and extensible compared to other widely used optimization frameworks.

Acknowledgements

We thank Jean-Paul Watson for feedback on the design of the PyUtilib Component Architecture. This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories. 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] B. A. Allan and et al., *A component architecture for high-performance scientific computing*, Intl J. of High Performance Computing Applications, 20 (2006), pp. 163–202.

[2] *Coopr: A common optimization repository.* `https://software.sandia.gov/trac/coopr`, 2010.

[3] *EnvisageCore.* `https://svn.enthought.com/enthought/wiki/EnvisageThree/core.html`, 2009.

[4] O. M. Group, *CORBA component model specification, version 4.0*, tech. rep., Object Management Group, Inc., 2006.

[5] G. Heineman and W. Councill, eds., *Component-Based Software Engineering, Putting the Pieces Together*, Addison-Wesley, 2001.

[6] R. Marvie, *Picolo: A simple python framework for introducing component principles*, in Euro Python Conference 2005, June 2005.

[7] J. C. Meza, *OPT++: An object-oriented class library for nonlinear optimization*, Tech. Rep. SAND94-8225, Sandia National Laboratories, 1994.

[8] *PyUtilib: A python utility library.* `http://software.sandia.gov/pyutilib`, 2009.

[9] *SprinklesPy.* `http://termie.pbworks.com/SprinklesPy`, 2009.

[10] *Trac component architecture.* `http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture`, 2009.

[11] *Trac.* `http://trac.edgewall.org/`, 2009.

[12] *yapsy.* `http://yapsy.sourceforge.net/`, 2009.

[13] *Zope.* `http://www.zope.org/`, 2009.