LHCb s/w week

18/3/2008

Job Configuration Using Python

Marco Clemencic

marco.clemencic@cern.ch

Overview

- Introduction
  - Why Pyhton configuration?
  - Status
- Tutorial
  - Foreword
  - Low level configuration
  - Extensions
  - High level configuration

Introduction

Some history

- September 2005 (Barcelona)
  Pere Mato introduced the idea.
  An implementation by Atlas, but not usable.
- September 2007
  Pere ported the more mature Atlas implementation to Gaudi and presented it at the Core Software Meeting.
- October 2007
  Status presented LHCb Software Week
- Now
  Bug fixes and improvements. Ready to be used.

Why Python configuration?

- Basic option validation
  (types, names, etc.)
- Programming language
  (loops, logic, modularization, etc.)
- High level configuration
  (simple command for complex configuration tasks)

Tutorial

Forewords/1

Some preliminary comments are needed to enter the correct state of mind:

- Job Options files are *configuration files*
  - C++-like syntax without checking
  - Simple parser
- Python is a OOP language
  - we have to deal with classes and objects
- Configuration **IS NOT** execution
  - do not confuse it with GaudiPython

Forewords/2

- The Python configuration is based on *Configurables*
- *Configurables* are special Python classes built from the C++ components (Services, Algorithms, Tools, . . .)
- Each Configurable instance has got a name that is unique by constructions

How to Run

Gaudi.exe is replaced by gaudirun.py:

```
> SetupProject Gaudi
Environment for Gaudi v19r8 ready.
(taken from /afs/cern.ch/sw/Gaudi/releases/GAUDI)
> gaudirun.py --help
Usage: gaudirun.py [options] <opts_file> ...

Options:
  -h, --help     show this help message and exit
  -n, --dry-run  do not run the application, just parse option files
  -p FILE, --pickle-output=FILE
                 write the parsed options as a pickle file (static
                 option file)
  -v, --verbose  print the parsed options
  --old-opts     format printed options in old option files style
  --all-opts     print all the option (even if equal to default)
```

Features of gaudirun.py

- Backward compatible
- Print the result of the configuration as Python dictionary or old options
- Parse the option files without running the application
- Dump the final configuration as a *pickle* file
- Accept more than 1 option file on the command line

Example

```
> SetupProject Brunel
Environment for Brunel v32r4 ready.
(taken from /afs/cern.ch/lhcb/software/releases/BRUNEL)
> gaudirun.py -v -n $BRUNELOPTS/v200601.py
...
# Dumping all configurables and properties (different from default)
{'ApplicationMgr': {'AppName': 'Brunel',
                    'AppVersion': 'v32r4',
                    'AuditAlgorithms': True,
                    'ExtSvc': ['DetDataSvc/DetectorDataSvc',
...
                               'AuditorSvc'],
                    'HistogramPersistency': 'ROOT',
                    'OutStream': ['DstWriter'],
                    'TopAlg': ['GaudiSequencer/BrunelSequencer']},
 'AuditorSvc': {'Auditors': ['TimingAuditor']},
...
```

Basic Configuration

Headers

Python needs to know which classes and functions you are going to use:

```
# import standard configurables and modules
from Gaudi.Configuration import *
# import non-standard configurables
from Configurables import MyAlgorithm, MyTool
```

- The first line should be present in all Python option files.
- The second line may not be needed.
  (Gaudi standard configurables like ApplicationMgr are always available)

Units

In .opts files it was possible to use units.
In Python you have to import the symbols from a special module:

```
# import all the units
from GaudiKernel.SystemOfUnits import *
```

or (better)

```
# import only the needed units
from GaudiKernel.SystemOfUnits import GeV, ns, ms
```

- Warning: importing all the units may conflict with your variables.

Basic usage

- Import (include) other option files

```
importOptions("myOldConf.opts")
importOptions("myNewConf.py")
importOptions("myPreparsedConf.pkl")
```

- Configure a service or an algorithm

```
alg1 = MyAlgorithm("MyAlg1")
alg1.AProperty = 1.234
svc1 = MySpecialSvc("MySvc")
svc1.OutputLevel = INFO
```

- Use the configured service or algorithm

```
app = ApplicationMgr()
app.TopAlg.append(alg1)
app.ExtSvc += [ svc1 ]
```

Tools

- Two flavors: public and private
- Configuration depends on the usage:
  - public
    1. `tool<IMyTool>("MyTool","TheTool")`
       `tool<IMyTool>("MyTool/TheTool")`
    2. `tool<IMyTool>(m_typename)`
    3. `tool<IMyTool>(m_type,m_name)`
    4. `tool<IMyTool>(m_type,"TheTool")`
  - private
    1. `tool<IMyTool>("MyTool","TheTool",this)`
       `tool<IMyTool>("MyTool/TheTool",this)`
    2. `tool<IMyTool>(m_typename,this)`
    3. `tool<IMyTool>(m_type,m_name,this)`
    4. `tool<IMyTool>(m_type,"TheTool",this)`
  - mixed

Public Tools: case 1

- `tool<IMyTool>("MyTool","TheTool")`
- `tool<IMyTool>("MyTool/TheTool")`

Old style:

```
ToolSvc.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
tool = MyTool("TheTool")
tool.PropertyOfTheTool = "ABC"
```

Not flexible: should be avoided.

Public Tools: case 2

- `tool<IMyTool>(m_typename)`

Old style:

```
MyAlg.ToolTypeName = "MyTool/TheTool";

ToolSvc.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
tool = MyTool("TheTool")
tool.PropertyOfTheTool = "ABC"

alg1.ToolTypeName = tool
```

Public Tools: case 3

- `tool<IMyTool>(m_type,m_name)`

Old style:

```
MyAlg.ToolType = "MyTool";
MyAlg.ToolName = "TheTool";

ToolSvc.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
tool = MyTool("TheTool")
tool.PropertyOfTheTool = "ABC"

alg1.ToolType = tool # 2nd property ignored
```

Public Tools: case 4

- `tool<IMyTool>(m_type,"TheTool")`

Old style:

```
MyAlg.ToolType = "MyTool";
// or
MyAlg.TheTool = "MyTool";

ToolSvc.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
tool = MyTool("JustAName") # the hardcoded name is ignored
tool.PropertyOfTheTool = "ABC"

alg1.ToolType = tool
```

Private Tools: case 1

- `tool<IMyTool>("MyTool","TheTool",this)`
- `tool<IMyTool>("MyTool/TheTool",this)`

Old style:

```
MyAlg.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
alg1.addTool(MyTool(),name = "TheTool")
alg1.TheTool.PropertyOfTheTool = "ABC"
```

Private Tools: case 2

- `tool<IMyTool>(m_typename,this)`

Old style:

```
MyAlg.ToolTypeName = "MyTool/TheTool";

MyAlg.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
alg1.addTool(MyTool(),name = "TheTool")
alg1.TheTool.PropertyOfTheTool = "ABC"
alg1.ToolTypeName = alg1.TheTool
# or
alg1.addTool(MyTool(),name = "ToolTypeName")
alg1.ToolTypeName.PropertyOfTheTool = "ABC"
```

Private Tools: case 3

- `tool<IMyTool>(m_type,m_name,this)`

Old style:

```
MyAlg.ToolType = "MyTool";
MyAlg.ToolName = "TheTool";

MyAlg.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
alg1.addTool(MyTool(),name = "TheTool")
alg1.TheTool.PropertyOfTheTool = "ABC"
alg1.ToolType = alg1.TheTool
# or
alg1.addTool(MyTool(),name = "ToolType")
alg1.ToolType.PropertyOfTheTool = "ABC"
```

Private Tools: case 4

- `tool<IMyTool>(m_type,"TheTool",this)`

Old style:

```
MyAlg.ToolType = "MyTool";
// or
MyAlg.TheTool = "MyTool";

MyAlg.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
alg1.addTool(MyTool(),name = "ToolType")
alg1.ToolType.PropertyOfTheTool = "ABC"
# or
alg1.addTool(MyTool(),name = "TheTool")
alg1.TheTool.PropertyOfTheTool = "ABC"
```

Forcing Public Tools

- `tool<IMyTool>(m_typename,this)`
  with `m_typename` ending with ":PUBLIC"

Old style:

```
MyAlg.ToolTypeName = "MyTool/TheTool:PUBLIC";

ToolSvc.TheTool.PropertyOfTheTool = "ABC";
```

Python style:

```
tool = MyTool("TheTool")
tool.PropertyOfTheTool = "ABC"

alg1.ToolTypeName = tool.getFullName() + ":PUBLIC"
```

Tools: Suggestions

- Public tools
  - use case 2: easier to handle

    `tool<IMyTool>(m_typename)`
- Private tools
  - case 2 is better
    (just give a good name to the property)

High Level Configuration

High Level Configuration

- Old .opts files
  - collection of files, one per use-case
  - files with a lot of includes, one per *super*-use-case (see Panoramix)
  - custom configuration: comment and uncomment lines
- Python
  - preconfigured components
  - functions
  - high level configurables

Preconfigured Components

It is possible to create a pool of pre-configured tools, algorithms or services. The user can choose which one of them to use.

- “preconfigured_analysis.py”

```
# Preconfigured algorithms
MyAlg("preconf_Selection1").Cut = 10 * MeV
MyAlg("preconf_Selection2").Cut = 15 * MeV
```

- “my_configuration.py”

```
importOptions("preconfigured_analysis.py")
ApplicationMgr().TopAlg.append(MyAlg("preconf_Selection1"))
```

Functions

Some configuration tasks need to manipulate the current configuration.

Practical example:

```
# utility function
from os.path import expandvars
# CondDB-related conf.  functions
from DetCond.Configuration import *
# configure local DB layer
mydb = CondDBAccessSvc("MyDB")
dbfile = expanvars("$HOME/tmp/myDB.db")
mydb.ConnectionString = "sqlite_file:%s/DDDB"%dbfile
# use the local DB
addCondDBLayer(mydb)
```

Functions (2)

To provide configuration functions you need to:

- put your functions in the file `python/<package>/Configuration.py`
- add to the requirements the line `apply_pattern install_python_modules`

See the example in the package Det/DetCond.

High Level Configurables

In some cases you need to collect informations to be used to prepare the actual low level configuration.

```
# just an example
LHCbApplication().CondDBTag = "DC06-repro0710"

# this does really work
from Boole.Configuration import *
Boole().useSpillover = False
Boole().writeRawMDF = True
```

Example of definition:

```
class LHCbApplication(ConfigurableUser):
      __slots__ = {}
```

Note: It is possible to write user-defined configurables, but the development is ongoing.

The future

Known issues

- Configuration of tools
  - nice idea and implementation at the beginning
  - a disaster if compared to the actual usage
- Backward compatibility
  - most of the know old options files work
  - the others should work 
- Use-cases coverage
  - all the use cases I know of are covered
  - if you have problems with your use case, just ask

Changes in the pipeline

- Improvements and bug-fixes
  - E.g. replace

    ```
    alg1.addTool(MyTool(),name = "ToolType")
    ```

    with

    ```
    alg1.addTool(MyTool,name = "ToolType")
    ```
- Implement automatic and generic handling of high level configurables

Thank you for your attention!