
Experience with Configurables

Juan Palacios

LHCb software week

October 2009

- The following all sounds very negative
- I've been concentrating on problems I've found developing new configurables
- Configurables are a great step forward from options files
- For most end users, they make life much, much easier
- For developers, they are much more powerful
- My issues are from the point of view of a developer who is trying to do things a bit differently
- So apologies for the negative tone!

- Author of DST writing configurables
 - Middle-level configurables (somewhere between algorithms, tools sequences and applications)
 - MicroDSTWriter, SelDSTWriter, BaseDSTWriter
- Author of Selection “framework”
 - Just wrappers around particle selection configurables and data-on-demand locations
- Hacker of high level configurables by others
 - DaVinci, PhysConf, AnalysisConf, StrippingConf
- Would-be author of hand-made DVAlgorithm base class configurable
 - But have no idea if it is possible

- Simple pattern -> should be easily re-usable
 - Take a list of DVAlgorithms
 - Add extra elements to the list
 - Return it to the user
 - Something like this:

```
conf0 = ConbineParticles(...)  
conf1 = FilterDesktop(...)  
checkPV = CheckPV("PVSafe")  
algs = [checkPV, conf0, conf1]  
from Configurables import DSTWriter, DaVinci  
dstWriter = DSTWriter("Writer0", Algos = algs)  
DaVinci().UserAlgorithms = dstWriter.sequence()
```

- Had some problems
 - Take a list of DVAlgorithms
 - Could not do this: the algorithms are missing parts of their configuration
 - Need to use a GaudiSequencer
 - Return it to the user
 - Also some issues here. Need to give back GaudiSequencer

- Questions to experts:
 - Should it have worked or not?
 - Why can't I make a list of DVAlgorithm configurables and pass it around between configurables?
 - GaudiSequencer seems to do something clever with its Members, can a lighter weight List configurable be made?

- Another problem:
 - DSTWriters do not depend on anything
 - Nevertheless, *when* their `__apply_configuration__` is called seems to depend on external factors
 - Presumably something is deciding some configuration parameter depends on something else that isn't defined yet? Or that could be re-defined?

- Weird case:
 - DSTWriter exports a GaudiSequencer that can be passed on to DaVinci() or others
 - See “sequence()” call in previous example
 - Recently, some change in a script made it return an empty sequence, for no apparent reason!

- Weird case:

This bit worked:

These lines make
`sc.activeStreams()`
`return []` so stripping
 Isn't run, DST isn't made.
 Why? Could it be because
 StrippingConf uses DaVinci?
 Why did it work before?

```

sc = StrippingConf(OutputType = "DST") # This gets modified later
dstWriter = SelDSTWriter("StripMC09DSTWriter",
                        SelectionSequences = sc.activeStreams(),
                        OutputPrefix = 'Strip', Extraltems = ['/Event/DAQ/
RawEvent'] )
DaVinci().DataType = "MC09" # Default is "MC09"
DaVinci().UserAlgorithms = [ dstWriter.sequence() ]
DaVinci().Hlt2Requires = 'L0+Hlt1'
DaVinci().HltType = 'Hlt1'
DaVinci().HltThresholdSettings = 'Charm_320Vis_300L0_10Hlt1_Aug09'
DaVinci().L0 = True
from Configurables import L0Conf,
L0DUFromRawAlg,
L0DUAlg L0Conf().TCK = '0xFF68'
L0Conf().DecodeL0DU = True
L0DUFromRawAlg().L0DUReportOnTES = False
importOptions("$L0DUROOT/options/L0Sequence.opts")
from Configurables import bankKiller
def appendL0DUAlg() :
removebanks=bankKiller( "RemoveL0Banks" )
removebanks.BankTypes = ["L0DU"]
l0du = L0DUAlg('L0DU')
l0du.DataLocations = ["Trig/L0/L0DUData"]
l0du.ReportLocation = "Trig/L0/L0DUReport"
l0du.StoreInBuffer = True
l0du.WriteOnTES = True
GaudiSequencer("seqL0").Members += [ removebanks, l0du ]
appendPostConfigAction(appendL0DUAlg)

```

- Another simple pattern: ***Selection***
 - Wraps DVAlgorithm configurable or data-on-demand location
 - Knows of other ***Selections*** it needs
 - Exports TES location to which it writes data
 - Sets **InputLocations** of its DVAlgorithm using the previous two points
 - Wanted it to be simple python class
 - But the DVAlgorithms don't seem to know about **InputLocations** property early enough
 - So I had to make it a ConfigurableUser and replace `__init__` by `__apply_configuration__`

- Another problem when using **Selection**
 - When **Selections** were being used by another configurable, information was lost because their `__apply_configuration__` hadn't been called.
 - I now call it manually each time I define a **Selection** configurable
 - Works fine, but what side-effects could it have?

- I've found many examples of behaviour when using configurables in a non-standard way
 - One example: GaudiSequencer Members being filled in exactly the opposite order to what you'd expect
 - Framework must be doing something clever here
- I admit my way of using them is probably not what they were defined for
 - But I think what I am trying to do makes more sense than the “standard” way of using configurables

- I don't like the fact that I can accidentally get a configurable made by someone else and change it's behaviour
 - Create-new and get-existing semantics should be different. Please!
- I want to be able to make "locked" configurables
 - If somebody wants to change something, they can clone it and modify the clone
- I do not even need to be able to get configurables by name.
 - Why does anybody need this? What's wrong with the python module system?
 - I am seriously running out of names for configurable instances!

- My experience with hacking other people's high level configurables
 - They are a mess of dependencies
 - Not clear what takes precedence over what
 - Single instances not enforced
 - I can make many DaVincis, but internally it uses the same instance of other things. This is not safe.
 - Configurables have behaved differently if I gave them a name
 - Because `__apply_configuration__()` was called at a different time

- Different create and get semantics
 - Trying to create something that already exists should result in error
- Possibility to lock a configurable upon construction
 - Although first point might be enough, since locking seems unpythonesque.
- List configurable to pass configurable instances between higher level configurable
- Enforcement on single-instance configurable
 - For example, for application configurables
 - How does the used_configurables scheme work if you can make different instances of PhysConf for example?

- Clear rules for when `__apply_configuration__()` gets called
 - If a configurable doesn't "use", it isn't "used" by another, then why doesn't it get configured immediately?
- Nameless configurables?
 - Instances defined in python module, not by name
 - Problem of having to define unique names all the time
- Recipe to make hand-made configurable for algorithms

- I have found it very hard to write simple, re-usable code patterns in configurables
 - I think others have too, hence the messyness of some high level configurables
- I think a lot of this is by design, but some might just be bugs
 - Is it trying to do too much?
 - Could some problems be solved by clone-and-change instead of accessing single instance by name and changing?
- I believe that configurables should be more like python and less like old style options
 - For example, one selling point was that one can use type checking, conditional statements and so on to control program flow. This turns out not to be true in many cases, since you cannot get a value for a property that doesn't exist yet.