Table of Contents

Stripping framework

Contents

Introduction

The current stripping framework is based on the code of Hlt2Lines and follows its concept.

Stripping framework consists of two packages belonging to the `ANALYSIS` project. `Phys/StrippingConf` contains definitions of python objects and C++ configurable. The actual selections and DaVinci jobs are contained in `Phys/StrippingSelections`.

Contrary to Hlt2Lines framework, no objects for shared particles definitions are created. It is expected that the states from CommonParticles will be used in the stripping.

Writing stripping selections

Selections are kept under `Phys/StrippingSelections/python/StrippingSelections/`. To add a new selection line, the python module with the `StrippingLine` constructor should be placed in this directory (with the name starting with `Stripping`). Possible ways of writing the selection are:

- Using plain Gaudi classes
- Using `StrippingMember`
- Using SelectionSequence or Selection

Examples are given below. In addition, the new selection has to be appended to any of the existing streams (or a new stream has to be created). Note that since DaVinci v24r4, the stream name is not a property of `StrippingLine`.

Since DaVinci v24r4, selections have been moved from options/ to python/. Therefore, do not forget to "cmt make" every time you change your selection!

Wrapping Selection object in `StrippingLine`

An existing Selection object can be used directly to create a `StrippingLine`. In the code below, `Jpsi` is a `Selection`. For details on how to build such an object, see the Particle Selection Framework page.

```
line = StrippingLine('JpsiInclusiveLine'
                     , prescale = 1.
                     , algos = [ Jpsi ] )
```

This will create a line with `Jpsi` 's algorithm and all the ones required by it. Only one `Selection` should be used for any given `StrippingLine`.

Wrapping offline selection sequencer in `StrippingLine`

A GaudiSequencer containing an old-style selection can be integrated directly into the stripping framework by using it to build a `StrippingLine` instance. In the following example, `B2DhFilterSequence` is a GaudiSequencer that has already been created:

```
line = StrippingLine('B2Dh'
               , prescale = 1
               , algos = [ B2DhFilterSequence ]
            )
```

This will create a line corresponding to `B2DhFilterSequence` selection, and add it to the list of lines.

Using `StrippingMember`

Using `StrippingMember` instead of common algorithm allows to clone `StrippingLine` in the same way as in Hlt2Line. This could be useful to handle e.g. signal and prescaled sidebands.

```
combine = CombineParticles("Bd2KstarMuMuCombine")

   filter = StrippingMember (FilterDesktop
                   , "Filter"
                   , InputLocations = ["Bd2KstarMuMuCombine"]
                   , Code = "{Some filter code}"
                 )

   line1 = StrippingLine('B2DPi_signal'
                   , prescale = 1
                   , algos = [ combine, filter ]
                 )

   line2 = line1.clone('B2Dpi_sideband'
                    , prescale = 0.2
                    , FilterDesktopFilter = { "Code" : "{Some looser filter code}" }
                 )
```

In the above example, `CombineParticles` will run only once, while two `FilterDesktop` instances will be created. This allows to modify one of the selection lines without affecting the other.

`StrippingLine` constructor arguments

The following options can be specified for a `StrippingLine`:

`algos = [Alg1, Alg2]`
: List of algorithms to run for this line. The algorithms can be `CombineParticles`, `FilterDesktop`, `SelectionSequence` or `Selection` instances, or `StrippingMembers`.

`prescale = 1`
`postscale = 1`
: Prescale and postscale factors. Prescaler and postscaler are run before and after the list of algorithms, respectively.

`checkPV`
: Controls checking of the existence of primary vertex in the event before running the sequence of algorithms. By default, checkPV=True, which means that at least 1 PV is required. Examples:
  - ◊ `CheckPV = False` - turn off PV existence check
  - ◊ `CheckPV = 2` - require at least 2 PVs
  - ◊ `CheckPV = (1,3)` - require from 1 to 3 PVs.

`FILTER = "filter string"`
: Check the result of VoidFilter before running the sequence of algorithm. Useful for applying global event cuts. Example:
  - ◊ `FILTER = "TrSOURCE('Rec/Track/Best') >> TrLONG >> (TrSIZE < 200 )"` - Require not more than 200 long tracks.

`ODIN = "ODIN predicate"`
: ODIN predicate

`L0DU = "L0DU predicate"`
: L0DU predicate

`HLT = "HLT predicate"`
: HLT predicate

`MaxCandidates = NN`
: Limit the number of candidates created by the line. If the number is exceeded, an incident "ExceedsCombinatoricsLimit" is triggered.

`MaxCombinations = NN`
: Limit the number of combinations in CombineParticles. If the number is exceeded, an incident "ExceedsCombinatoricsLimit" is triggered.

Testing that additions or modifications do not break the standard stripping

If any modifications or additions have been made to the StrippingSelections module, it is mandatory to check that these do not break the instantiation of the stripping. This can be achieved by running the QMTests, after setting the DaVinci environment and building the package. From `Phys/StrippingSelections/cmt`:

```
cmt qmtest_run
```

Running stripping job

Configuring the stripping: `StrippingConf` options

The object `StrippingConf` takes care about configuration of the stripping job:

```
from StrippingConf.Configuration import StrippingConf
```

This object creates and holds a list of StrippingStreams and contains member functions to access sequencers, output locations and selection names needed to be passed to DaVinci and DST writers. It is simple to configure, and new streams can be defined and added to it. Here, we construct a `StrippingConf` with four official stripping selections:

```
from Gaudi.Configuration import *
from StrippingConf.Configuration import StrippingConf

# import some official Stripping Selections
from StrippingSelections import StreamBmuon, StreamHadron, StreamJpsi, StreamDstar

sc = StrippingConf(Streams = [StreamBmuon.stream,
                  StreamHadron.stream,
                  StreamJpsi.stream,
                  StreamDstar.stream] )
```

It is also possible to add streams to an existing `StrippingConf` instance, using either the `appendStream` or the `appendStreams` method.

```
from StrippingSelections import StreamLambda, StreamBelectron
sc.appendStream( StreamLambda.stream )
sc.appendStream( StreamBelectron.stream )
```

Note that since DaVinci v24r4, `StrippingConf` does not configure DaVinci to write ETC or DST files internally, this is done in job options. Once the `StrippingConf` is ready, it can be used to simply run a job, write an ETC, or write some stripping DSTs, as the following examples show.

`ETC` output

This example of ETC output job is from `Phys/StrippingSelections/tests/TestStrippingETC.py`. It assumes that a `StrippingConf sc` has been instantiated and configured as shown above.

```
from Configurables import EventTuple, TupleToolSelResults

tag = EventTuple("TagCreator")
tag.EvtColsProduce = True
tag.ToolList = [ "TupleToolEventInfo", "TupleToolRecoStats", "TupleToolSelResults"  ]
tag.addTool(TupleToolSelResults)

tag.TupleToolSelResults.Selections = sc.selections()  # Add the list of stripping selections to T

from Configurables import DaVinci

DaVinci().appendToMainSequence( [ sc.sequence() ] )   # Append the stripping selection sequence t
DaVinci().appendToMainSequence( [ tag ] )             # Append the TagCreator to DaVinci
DaVinci().EvtMax = 1000                        # Number of events
DaVinci().ETCFile = "etc.root"                 # ETC file name

importOptions("$STRIPPINGSELECTIONSROOT/tests/MC09_Bincl.py")     # Data file
```

`DST` output

The `StrippingConf` object and streams are configured in the same way as for ETC writing. After that, one has to configure SelDSTWriter and pass it to DaVinci:

```
dstWriter = SelDSTWriter("MyDSTWriter",
        SelectionSequences = sc.activeStreams(),
        OutputPrefix = 'Strip',
        OutputFileSuffix = '000000'
        )

DaVinci().EvtMax = 1000                         # Number of events
DaVinci().UserAlgorithms = [ dstWriter.sequence() ]
```

Testing single streams or lines

The above jobs use all streams available in `sc`. To test individual selections, one can create the test stream, e.g. to test only the `StrippingB2Dh` line, one can do:

```
from StrippingConf.StrippingStream import StrippingStream
from StrippingSelections import StrippingB2Dh

stream = StrippingStream("Test")

stream.appendLines( [ StrippingB2Dh.line ] )

from StrippingConf.Configuration import StrippingConf

sc = StrippingConf(Streams=[stream] )

...
```

Monitoring algorithm

An algorithm to monitor the selection results called `StrippingReport` is available. It can provide the following information:

- Selection results in each event
- Summary of selection results (number of selected events, accept rate and average multiplicity) every N events.
- Summary of selection results in the end of the job.
- List of noisy selections (with the accept rate above certain threshold) and non-responding selections (that select zero events)

The algorithm has to be initialised with the list of selection names, e.g. for all configured lines:

```
from Configurables import StrippingReport
sr = StrippingReport(Selections = sc.selections());
```

and added to the end of the list of DaVinci algorithms:

```
DaVinci().appendToMainSequence( [ sr ] )
```

The following properties are available:

- `OnlyPositive` (default: `True`) - show only the selections with at least one event accepted.
- `EveryEvent` (default: `False`) - show selection results for every event
- `ReportFrequency` (default: 100) - number of events between selection summaries during the job execution
- `PrintNonResponding` (default: `True`) - list non-responding selections (the ones that select zero events) at the end of the job
- `PrintHot` (default: `True`) - list "hot" selections (with accept rate above threshold defined in `HotThreshold`) at the end of the job
- `HotThreshold` (default: 0.5) - accept rate threshold for "hot" selections.
- `Latex` (default: False) - if True, use Latex table style instead of TWiki style by default.

To get the timing information in `StrippingReport` you need to add `ChronoAuditor` service:

```
# ChronoAuditor is used by StrippingReport to show the timing.
from Configurables import AuditorSvc, ChronoAuditor
AuditorSvc().Auditors.append( ChronoAuditor("Chrono") )
```

Isolation Tools in the Stripping

See the StrippingIsolationTools TWiki page.

-- JuanPalacios - 26-Feb-2010

This topic: LHCb > StrippingLines
Topic revision: r37 - 2014-07-22 - AlexShires