Table of Contents

Generic Selectors and Filter Modules

Complete:

Introduction

Please note that the things described here can't be used in the SWGuideHighLevelTrigger code of CMS.

Generic Filter Modules

Generic Filter Modules are designed to simplify the task of writing event filter modules based on the selection of a number of objects passing some simple selection criteria.

One example of such criteria could be:

- *"filter events with at least two electrons with $p_t > 5$ GeV/c in the range $-1.5 < \quad < 1.5$"*

Generic Selectors

Selecting objects in a collection according to user-defined criteria is a common requirement on CMS applications. Usually, selectors could be more complex than filter modules.

The ***Generic Selector*** toolkit under the Physics Tools system provides a flexible and configurable way to write custom selector modules.

Examples of user requirements could be:

- *"select tracks with $p_t > 10$ GeV/c"*
- *"select tracks within a cone of aperture $\Delta R < 0.4$ around the two most energetic electrons"*
- *"select electrons pairs with an invariant mass close to the Z mass within 5 GeV/c^2"*
- etc., etc.

A generic selector producer module should perform the following tasks:

1. get a source collection of a specified type from the event
2. select the objects in the collection passing a specified sets of requirements
3. store a new collection of objects of the same type in the event containg only clones of the selected objects

Generic selectors could also behave as event filter modules, filtering events only if at leas one element in the specified collection is selected.

There are two level of generality required for a selector module:

1. the user should specify the selection in a fiexible and generic way without caring about the technicalities related with cloning the selected objects
2. the system should perform correcly the cloning operation of the selected objects. Those could be non trivial. For instance, in order to clone a set of selected tracks, three branches must be cloned: **Tracks**, **TrackExtras**, **TrackRecHits**, and the proper references among those objects must be set. This should be possible in general, for all types of object types that need to be selected.

Generic Single-Object Selectors

The user can specify his selection criteria on a specific object type defining a functor class, similarily to what is used in many STL applications.

A simple selector could be the following:

```
struct PtMinTrackSelector {
  PtMinSelector( double ptMin ) : ptMin_( ptMin ) { }
  PtMinSelector( const edm::ParameterSet & cfg ) :
    ptMin_( cfg.template getParameter<double>( "ptMin" ) ) { }
  bool operator()( const reco::Track & t ) const { return t.pt() > ptMin_; }
private:
  double ptMin_;
};
```

The selector should define two ingredients:

1. the operator "()", taking one object as argument, returning true if the object passes the selection, false otherwise
2. define the typedef `value_type` specifying the type used as argument of the "()" operator

In this way, the selector can be used as a function, according to the following example:

```
PtMinTrackSelector select( 15.0 );

const reco::Traco & track = . . . // get a track
if ( select(  track ) ) { . . . }
```

Several already defined selectors are defined in genric way in the package:

- `CMS.CommonTools/Utils`

and can be used on any object type that define common member function (e.g.: `pt()`, `et()`, `eta()`, ...). Some of those are:

- `PtMinSelector`: selects objects with p_t greather than the cut `ptMin`;
- `EtMinSelector`: selects objects with E_t greather than the cut `etMin`;
- `EtaRangeSelector`: selects object with *eta* within the range specified by the parameters `etaMin` and `etaMax`.
- `ChargeSelector`: selects object with *charge* having a value specified by the parameter `charge`.
- `MassRangeSelector`: selects object with *mass* within the range specified by the parameters `massMin` and `massMax`.
- `PdgIdSelector`: selects object with *pdgId* in a vector of values specified by the parameter `pdgId`.

Combining Single Object Selectors

Selector can be combined using logical AND and OR. This is provided by the utility selectors `AndSelector` and `OrSelector` defined in the package:

- `CMS.PhysicsTools/Utilities`

For instance, to select objects within a given   range and with p_t above a given cut, the following selector can be used:

```
typedef AndSelector<
```

```
     EtaRangeSelector,
     PtMinSelector
    > EtaPtMinSelector;
```

Generic Object Pair Selectors

The user can specify his selection criteria on a specific object type defining a binary functor class, similarily to what is used in many STL applications.

A simple example could be the following:

```
struct NeutralChargeSelector {
  NeutralChargeSelector ( const edm::ParameterSet & cfg )  { }
  template<typename T1, typename T2>
  bool operator()( const T1 & t1, const T2 & t2 ) const {
    return t1.charge() + t2.charge() == 0;
  }
private:
  double ptMin_;
};
```

A few generic object pair selectors are defined in the package:

- `CMS.PhysicsTools/Utilities`

and are based on a user-provided binary functor class that returns a value given a pair of objects (could be invariant mass, ΔR, Δ etc.):

- RangeObjectPairSelector: selects a pair of objecs if a specific variable lies within a specified range;
- MinObjectPairSelector: selects a pair of objecs if a specific variable is larger than a specified lower bound;
- MaxbjectPairSelector: selects a pair of objecs if a specific variable is larger than a specified upper bound;

Generic Object Selector and User-Defined Selections

The most general selector producer type is the template: **`ObjectSelector<S>`**. The user should specify its selection defining a custom class that should be passed in place of the template argument **`S`**. The structure of the class should be the following:

```
class MySelection {
  // collection type
  typedef [...] collection;
  // iterator over result collection type.
  // could be std::vector<const collection::value_type *>::const_iterator;
  typedef [...] const_iterator;
  // constructor from parameter set configurability
  MySelection( const edm::ParameterSet & );
  // select object from a collection and
  // possibly event content
  void select( const collection &, const edm::Event & );
  // iterators over selected objects: collection begin
  const_iterator begin() const;
  // iterators over selected objects: collection end
  const_iterator end() const;
  // true if no object has been selected
  size_t size() const;
};
```

Warning: from release 2.0.0 on, the function `select` will also take the `EventSetup` as argument:

```
 void select(const collection &, const edm::Event &, const edm::EventSetup &);
```

Notice that the `select` function receives the `edm::Event` as argument, so the selection can be completely generic. Also, the constructor receives the module's ParameterSet, so all selection cuts can become module parametrs.

The module also works as a Filter Module, and select an event only if at least one object has been selected.

An simple example of user-defined class is be the following, that select all tracks with p_t greater than the specified parameter `ptMin`:

```
struct MySelector {
  typedef reco::TrackCollection collection;
  typedef std::vector<const reco::Track *> container;
  typedef container::const_iterator const_iterator;
  MySelector ( const edm::ParameterSet & cfg ) :
    ptMin_( cfg.getParameter<double>( "ptMin" ) ) { }
  const_iterator begin() const { return selected_.begin(); }
  const_iterator end() const { return selected_.end(); }
  void select( const collection & c, const edm::Event & ) {
    selected_.clear();
    for( reco::TrackCollection::const_iterator trk = c.begin();
         trk != c.end(); ++ trk )
      if ( trk->pt() > ptMin_ ) selected_.push_back( & * trk );
  }
  size_t size() const { return selected_.size(); }
private:
  container selected_;
  double ptMin_;
};
```

In order to declare the producer module that uses the above selection, the user needs to use the following:

```
#include "PluginManager/ModuleDef.h"
#include "FWCore/Framework/interface/MakerMacros.h"

// include the definition of custom cloning procedure
// for track collection, that clones together with Tracks,
// also TrackExtras and RecHits
#include "CMS.PhysicsTools/RecoAlgos/interface/TrackSelector.h"

// include the definition of you selection
#include ".../MyPackage/inteface/MySelector.h"

// define your producer name
typedef ObjectSelector<MySelector> MyTrackSelector;

// declare the module as plugin
DEFINE_FWK_MODULE( MyTrackSelector );
```

Selector Utilities for Simplified Selections

Some of the possible selection criteria could be simplified w.r.t. the most general case, and utilities are provided to simplify the definition of the selection class in some of those cases. ***The set of supported selections could be expanded according to user requests***.

Single Element Collection Selection

Can be used to specify selection of single objects in a collecition passing specified criteria that do not need the rest of the events or other objects in a collection. For instance, inorder to select all tracks with p_t above a given threshold, it is sufficient to specify the following module type:

```
typedef ObjectSelector<
            SingleElementCollectionSelector<
              reco::TrackCollection,
              PtMinSelector
            >
          > PtMinTrackSelector;
```

This is equivalent to the recommended simpler syntax:

```
typedef SingleObjectSelector<
              reco::TrackCollection,
              PtMinSelector
          > PtMinTrackSelector;
```

Any other other single object selector could be used, provided that:

1. defines a constructor from a ParameterSet
2. behaves as a functor, i.e. defines the operator `()` with the object type as argument.

Sort Collection Selection

Select the first **N** objects (**N** being configurable via the parameter `max`), according to a specified sorting criterion. For instance, to select the **N** tracks having the largest p_t, the following module can be specified:

```
typedef ObjectSelector<
            SortCollectionSelector<
              reco::TrackCollection,
              PtInverseComparator<reco::Track>
            >
          > LargestPtTrackSelector;
```

Where PtInverseComparator and PtComparator can be taken from:

- `CMS.PhysicsTools/Utilities`

Any other other comparator could be used, provided that it defines a boolean operator "`()`" taking two (references to) objects in the selected collection as argument.

Window Collection Selector

Selects object pair whose property (could be invariant mass, ΔR, Δ or anything) lie within a specified range. The range is specifies with the parameters `min` and `max`. For instance, to select track pairs with invariant mass within a specified range, it is possible to use the following module:

```
typedef ObjectSelector<
            WindowCollectionSelector<
              reco::TrackCollection,
              MasslessInvariantMass
            >
          > MassWindowTrackSelector;
```

Where MasslessInvariantMass is taken from:

- `CMS.PhysicsTools/Utilities`

String-based custom selection

A flexible single-object selection can be done with a simple string that is parsed by the =StringCutObjectSelector= utility. For instance, a configurable track selector can be declared as follows:

```
typedef ObjectSelector<
          SingleElementCollectionSelector<
            reco::TrackCollection,
            StringCutObjectSelector<reco::Track>
          >
        > ConfigTrackSelector;
```

Tracks with pt greater than 15 GeV/c can be selected with the configuration below:

```
module bestTracks = TrackSelector {
  InputTag src = ctfAnalyticalTracks
  string cut = "pt > 15.0"
}
```

variable names (like `pt`) are mapped to object methods via Reflex dictionary. The cuts can be set using different comparators, algebraic adn boolean operators and the most commonly used math functions. More information on:

- Physics Cut Parser.

Caveat about string cuts

using string cuts can be dangerous. The module's parameters are saved as part of the product provenance. Two modules with equivalent cuts could have different string cuts (e.g.: **`"m > 1.0"`** or **`"m > 1"`** or adding/removing blanks: **`"m>1"`**, etc.) that would result in different product identification. When using string-based cuts, such product identification could become unreliable.

Output collection types

Generic selection can also save collections of references to selected objects. This allows to save disk space and runs faster than cloning selected objects. It is possible to specify a output collection type an `edm::RefVector<C>` as extra template parameter, as it is done below:

```
typedef SingleObjectSelector <
            reco::CandidateCollection,
            PdgIdSelector,
            reco::CandidateRefVector
        > PdgIdCandRefSelector;
```

or

```
typedef ObjectSelector<MySelector,reco::CandidateRefVector> MyCandRefSelector;
```

N.B. In the latter case, the functions begin() and end() in the `MySelector` class that you supply (see here) must point to a container of type `RefVector`.

From release 1_5_0, for input collection of type:

- `edm::AssociationVector<R, C>`

corresponding output collections of the following types are supported:

- `R::product_type`
- `edm::RefVector<R::product_type>`

For instance, an input of type:

- `edm::AssociationVector<edm::RefProd<reco::MuonCollection>, std::vector<float> >`

it is possible to save muon clones in an `reco::MuonCollection` or references to the selected muons in a `edm::RefVector<reco::MuonCollection>`.

This may be useful to select, for instance, isolated muons according to a configurable isolation cut:

```
typedef SingleObjectSelector<
          edm::AssociationVector<reco::MuonRefProd, std::vector<float> >,
          PairSelector<
            RefSelector<AnySelector>,
            MaxSelector<float>
          >,
          reco::MuonRefVector
        > IsolatedMuonRefVectorSelector;
```

Used as:

```
  module isolatedMuons = IsolatedMuonRefVectorSelector {
    InputTag src = muonIsolations  # tag of association vector containing isolations
    double max = 0.3 # cut on isolation
  }
```

The utilities `PairSelector<R, T>` and `RefSelector<T>` are defined in `CMS.PhysicsTools/Utilities`, and allow to match the `AssociationVector<R, C>::value_type` to the required selector inputs.

Object Counting Filters

A filter module that filters events with at least one or **N** objects passing a user defined selection can be specified using the tamplate `ObjectCountFilter<C, S>` where `C` is the collection type used to filter events, `S` is the single object selector type.

For instance, a module can be put in the `SealModule.cc` file, in the `src` directory of your favorite package, as in the following example:

```
#include "PluginManager/ModuleDef.h"
#include "FWCore/Framework/interface/MakerMacros.h"

// include the definition of you selection
#include "CMS.PhysicsTools/Utilities/inteface/PtMinTrackSelector.h"

// define your producer name
typedef ObjectCountFilter<
             reco::TrackCollection,
             PtMinSelector
           > PtMinTrackCountFilter;
```

```
// declare the module as plugin
DEFINE_FWK_MODULE( PtMinTrackCountFilter );
```

The filter module configuration will have two parameters:

1. `minNumber`, the minumum number of objects passing the selection required to filter the event, required by `ObjectCountFilter`. If not specified the value one is taken as default;
2. `ptMin`, the parameter required by `PtMinSelector`.

So, the module configuration could be:

```
module trackFilter = PtMinTrackCountFilter {
  uint32 minNumber = 2
  double ptMin = 5.0
}
```

Object Pair Filters

A filter module that filters events with at least one or **N** object pairs passing a user defined using the tamplate `ObjectPairFilter<C, S>` where `C` is the collection type used to filter events, `S` is the object pair selector type.

For instance, a filter based on the invariant mass of muon pairs can be specified as follows:

```
    typedef ObjectPairFilter<
              reco::MuonCollection,
              RangeObjectPairSelector<
                reco::Muon,
                MasslessInvariantMass
              >
            > MuonPairMassFilter;
```

Other Generic Utilities

Other generic utilities are supported, more can be added in the future.

Merging Uniform Collections

Collections of the same type can be merged, the output being a single collection of the same type containing clones of the input collections. A generic producer module `Merger<C>`, `C` being the collection type is provided for this purpose. For instance, the following module merges collections of electrons:

```
  typedef Merger<reco::ElectronsCollection> ElectronMerger;
```

And can be used according to the following configuration:

```
  module allElectrons= ElectronsMerger {
    VInputTag src = { electronsA, electronsB }
  }
```

Review status

Reviewer/Editor and Date (copy from screen)	Comments
LucaLista - 30 Aug 2006	Created topic
AnneHeavey - 15 Sep 2006	moved topic to workbook
LucaLista - 28 Sep 2006	added generic filter modules

Responsible: LucaLista
Last reviewed by: LucaLista - 30 Aug 2006

This topic: CMSPublic > SWGuideGenericSelectors
Topic revision: r35 - 2015-11-09 - IvanRazumov