Design of a Cut Recorder for TRF

July 3, 2000

Contents

1 Requirements

The purpose of this document is to specify a design for a set of data structures and classes for recording cuts in TRF. The requirements are as follows:

- A cut information record will be optionally generated and saved for each candidate track or collection of clusters considered by TRF for which a cut can be applied, whether or not the candidate track passes or fails the cut.

- Each cut information record will contain the following information:
  - A unique candidate track identifier.
  - The identifier of the parent track.
  - The TRF path id.
  - A flag indicating whether or not the candidate track passed or failed its cuts.
  - A flag indicating whether this candidate track is included in the final sample of found tracks.
- Each cut for which data is recorded will be associated with a unique string name.
- It will be possible to turn recording on or off globally, or for a specified subset of cuts.
- Implementation will be such that memory and cpu time overheads are negligible if cut recording is turned off.

2 New Classes

The following new classes will be created:

CutRecord Holds cut information for each candidate track.

CutRecorder Store `CutRecord`'s for one tracking step.

CutID A unique identifier for each cut. `CutID`'s will be associated with the string names through a cut registry. The cut registry will be part of `CutRecorder`.

These new classes will be stored in a new cvs library called `trfcut`. Cvs library `trfcut` will also contain classes to analyze the collected data and generate, for example, an ntuple. However, at this time, this document only specifies the design of the above data collection classes.

2.1 Class CutID

Class `CutID` will have an `int` as its single attribute. A unique `CutID` will be associated with each cut, as identified by a unique string name. The association between `CutID`'s and cut names will be maintained by a cut registry, which is part of class `CutRecorder` (see below). `CutID` will have a method to determine whether an instance is valid.

2.2 Clsss CutRecord

Class `CutRecord` will hold information relating to cuts applied to each candidate track. Instances of `CutRecord` will exist in roughly one-to-one correspondence with TRF `PTrack`'s (there is an exception to this rule when cuts that are applied at the cluster level, before instantiation of a `PTrack`). `CutRecorder` will hold the following attributes:

- A unique integer candidate track identifier or `PTrack` identifier.
- The identifier of the parent track.
- An integer TRF path id.
- A boolean flag indicating whether the track passed all of its cuts.
- A boolean flag indicating whether this track is part of the final sample of good tracks.
- A collection of cut values, implemented as STL `map<CutID, double>`. The collection must be of type `map` because any given instance of `CutRecord` will likely store only a small subset of the enabled cuts.

2.3 Class CutRecorder

Class `CutRecorder` will be implemented as a singleton, so that it is accessible from anywhere in a TRF program. The `CutRecorder` will contain a cut registry and a collection of `CutRecords`.

2.3.1 Cut Registry

.

The cut registry in class `CutRecorder` will consist of a collection of type `map<string, CutID>` representing a mapping from cut names to valid `CutID`'s. The cut registry will be populated at package initialization time under RCP control from `GtrFindStepper`. It may also be useful to add the reverse mapping, but for the data collection phase only the forward mapping is necessary. Once the cut registry is populated, it remains unchanged for the life of the program.

2.3.2 Cut Record Collection

The cut record colleciton in class `CutRecorder` will be of type `vector<CutRecord>`, where the vector index will correspond to the track id. There is the possibility of having to store `CutRecord`'s that do not have a valid track id's. These will require a second STL collection. The cut record collection will be cleared at the beginning of each tracking step in `GtrFindStepper`, and analyzed or otherwise processed at the conclusion of each tracking step. Thus, the cut record collection does not need to remain valid outside of a single invocation of `GtrFindStepper::process`, thus multiple tracking steps can share the single CutRecorder.

2.4 Life cycle of a CutRecord

All `CutRecord`'s are owned and managed by the `CutRecorder` singleton. `CutRecord`'s should never be constructed directly outside of `CutRecorder` (the constructor(s) of `CutRecord` will probably be private). It will be possible for users of `CutRecord`'s to generate new ones (i.e. add a new `CutRecord` to the cut record collection in `CutRecorder`) by invoking methods of `CutRecorder` or `CutRecord`. The following methods are envisioned:

```
CutRecord* CutRecorder::make_cut_record(int id, int parent_id, int path_id);
CutRecord* CutRecord::make_cut_record(int id, int path_id);
```

Either method returns a pointer to a `CutRecord` in the cut record colleciton inside `CutRecorder`. The first method is a general purpose method for generating a `CutRecord` with the specified id.'s. The second method is a short cut for generating a cut record which is a child of a given parent cut record. If cut recording is globally disabled, these methods return a null pointer.

The main place where new `CutRecord`'s will be generated is in the `PTrack` constructor. An attribute corresponding to a `CutRecord` will be added to `PTrack`, which will be initialized whenever a new `PTrack` is constructed (unless the cut recording mechanism is globally turned off). A pointer to the `CutRecord` (which can be null) will be passed to track cutters (checkers, fitters, and possibly filters), which will then be expected to add their cut data to the `CutRecord`.

Once a new `CutRecord` is generated and added to the cut record collection, it will remain in the collection until the collection is cleared at the beginning of the next tracking step.

3 Modifications to Existing Classes

Existing TRF and GTR classes will need to be modified to interact with the cut recording mechanism. The required changes are summarized in this section.

3.1 GtrFindStepper

Class `GtrFindStepper` represents the top level of the cut recording mechanism.

The cut registry will need to be initialized in the constructor of `GtrFindStepper` prior to instantiation of `objstream` objects. Based on the step RCP parameters, `GtrFindStepper` will need to determine whether cut recording is enabled for this step (other classes can query the global initialization flag through the `CutRecorder`), and add to the cut registry any cuts for which data is to be recorded. The names of the recorded cuts will be specified in the step RCP file.

During event processing, the global recording flag will be set and the cut record collection will be cleared at the beginning of each tracking step in `GtrFindStepper::process`.

3.2 TrackExtender

The most logical place to analyze data collected by the `CutRecorder` is at the conclusion of track finding in `TrackExtender::extend`.

3.3 PTrack

Class `PTrack` will interact extensively with the cut recording mechanism. Here is a list of required changes.

- Class `PTrack` will get a new attribute of type `CutRecorder*`. This attribute will point to the `CutRecord` inside the `CutRecorder` that corresponds to the `PTrack` instance. If cut recording is turned off globally, this pointer will be null. In order to properly account for parent-child relationships among `CutRecords`, it will be necessary to add an argument of type `CutRecord*` to the full initializing constructor of `PTrack`, which will point to the parent `CutRecord`.

- At place in `PTrack::propagate` where new tracks are constructed, these constructor calls will need to get an extra argument of type `CutRecord*`, which will enable the `PTrack` constructor to properly initialize its `CutRecord*` attribute.

- In `PTrack::propagate` where cutting methods are invoked (e.g. checkers, fitters, cluster filters, and possibly filters), these calls will also have to be modified to include an extra argument of type `CutRecord*`.

- Cluster filters will need to get an extra invocation after construction of new `PTracks` but before fitting.

3.4 Checkers

Concrete checkers derive from abstract base class `Checker`. Concrete checkers are currently required to override pure virtual method `Checker::status(const MTrack&)`.

3.4.1 Base Class Checker

.

Base class `Checker` will be modified as follows:

- A new virtual method having the following signature will be added:

  ```
  virtual bool status(const MTrack&, CutRecord*);
  ```

  This method is the same as the existing pure virtual method except for the addition of a new `CutRecord*` argument. This method will be given a default implementation which calls the old `status` method (i.e. the `CutRecord` argument is ignored).

- Instead of being pure virtual, the old `status` method will be given a default implementation that crashes the program.

`PTrack::propagate` will call the new version of `status`. If cut recording is turned off globally, the new `CutRecord*` argument will be a null pointer.

3.4.2 Concrete Checkers

Concrete checkers will be allowed to override either or both of the `status` methods. Thus, the new interface is backward compatible with existing concrete checkers.

Checkers that wish to implement the cut recording feature should to the following:

- In their constructor, checkers should query the cut registry to obtain a `CutID` for each cut that they wish to record. The recorder name of the cut or cuts implemented by a checker could be hard-coded, or obtained from an `.obs` file, in case there is more than one instance of a given checker.

- In their status method, each checker should invoke method `add_cut` of the passed in `CutRecord`. This method has the following signature:

  ```
  void add_cut(const CutID& id, double value);
  ```

- Method `status` should check for a null pointer to the `CutRecord`, and for valid `CutID`'s (`CutRecord::add_cut` also checks the `CutID` for validity).

Concrete checkers that do not wish to implement the cut recording feature do not need to do anything. However, since the goal is to record every TRF cut, concrete checkers should aim to implement the recording feature.

3.5 AddFitters

Concrete fitters derive from base class `AddFitter`, which provides two virtual methods which can be overridden called `add_hit` and `add_hit_fit`. `PTrack::propagate` calls method `add_hit`, which has a default implementation that calls `add_hit_fit`. Concrete fitters are allowed to override either `add_hit` or `add_hit_fit`.

3.5.1 Base Class AddFitter

.

Base class `AddFitter` will be modified similarly to `Checker`. New versions of `add_hit` and `add_hit_fit` will be added which have an extra argument of type `CutRecord*`. `PTrack::propagate` will call the new version of `add_hit`. The new versions of `add_hit` and `add_hit_fit` will get default implementations that call the old versions of the same method. Note that this is not the only reasonable default implementation of `add_hit`. The other reasonable default implementation would have been one that called the new version of `add_hit_fit` instead of the old version of `add_hit`. However, this default implementation `add_hit` is the one that gives backward compatibility.

3.5.2 Concrete AddFitters

Concrete fitters that wish to implement the recording feature should do the following:

- In their constructor, fitters should query the cut registry to obtain a `CutID` for each cut that they wish to record. The recorder name of the cut or cuts implemented by a fitter could be hard-coded, or obtained from an `.obs` file.
- Concrete fitters should invoke method `add_cut` of `CutRecord` in their `add_hit` or `add_hit_fit` method.
- As with checkers, fitters should check for a null pointer to the `CutRecord`, and for valid `CutID`'s.
- Because of the backward compatible default implementation of the new version of `add_hit`, concrete fitters that wish to implement cut recording *must* override method `add_hit`, even of they also override `add_hit_fit`. If there are many concrete fitters in this situation, an intermediate level base class could be provided with the other default implementation for the new version of `add_hit`.

As with checkers, concrete fitters that do not wish to implement the cut recording feature do not need to do anything. Unlike checkers, fitter do not inherently need to make cuts, so it may be reasonable to leave many existing fitters alone to operate in backward compatibility mode, or to move their cuts into `ClusterFilters`.

3.6 Cluster Filters

Concrete cluster filters derive from base class `ClusterFilter` and are required to override pure virtual method `filter`. The situation with cluster filters is more complicated than it is for either checkers or fitters, since normally, cluster filters are called before a candidate `PTrack` has been generated. The complications are explained futher in the following section.

3.6.1 Base Class ClusterFilter

Base class `ClusterFilter` will be modified as follows:

- A new version of method `filter` will be added having the following signature:

  ```
  virtual void filter(const MTrack&, ClusterList&, CutRecord*) const
  ```

  This method is the same as the existing `filter` method with an extra argument of type `CutRecord*`. This method will have a default implementation that calls the old `filter` method.

- A second new version of `filter` will be added with the following signature:

  ```
  virtual void filter(const MTrack&, CutRecord*) const
  ```

This version drops the `ClusterList` argument (the new cluster is assumed to be part of the track). This method will have a default implementation that does nothing.

- Instead of being pure virtual, the old `filter` method will be given a default implementaion that crashes the program.

`PTrack::propagate` will be modified to call version 2 of method `filter` before new candidate tracks are generated (where it does now), and then it will call version 3 of method `filter` after generating candidate tracks, but before fitting them. The latter call can be omitted if cut recording is turned of globally.

3.6.2 Concrete ClusterFilters

Concrete cluster filters that do not wish to implement cut recording do not need to change anything (but they should, as the do cut tracks). Concrete clusters filters that wish to implement cut recording should do the following:

- In their constructor, cluster filters should query the cut registry to obtain a `CutID` for each cut that they wish to record.
- Concrete cluster filters should override version 2 and version 3 of method `filter`.
- In version 2 of method `filter`, for each cluster *that fails the cut*, a new `CutRecord` should be generated from the parent `CutRecord` (i.e. the one that is passed in) and a cut value calculated. The new `CutRecord` should be given a track id. of -1, since the failed clusters to not correspond to any `PTrack`.
- In version 3 of method `filter`, a cut value should be calculated and added to the `CutRecord` from the clusters in the track that is passed in. In this method, it should be a fatal error for any track to fail the cut, since bad clusters should have already been eliminated. Note that tracks passed to version 3 of method `filter` should have one more cluster than tracks passed to version 2 of method `filter`.