This is a proposal for a more maintainable and effective structure of the software of the HLT alleys. It consists of three different parts: a **base class** for the algorithms executed in the alleys, a couple of **common algorithms** that can be shared between alleys, and a list of **additional guidelines** for code to be written in specific algorithms.

Most of the code needed for this proposal has in fact already been developed for the use of the Hadron and Muon+Hadron alleys, mostly by reusing code from the rest of the alleys, hence it could be made available for trying next week.

If we agree in this proposal, this document with the corresponding modifications can evolve into a 'style manual' for writing the code for the alleys.

1. HltAlleyAlgo base class

The aim is that the base class performs all the actions that are expected to be common for all algorithms in the alleys. Some of the advantages of this approach are:

- remove opportunities of typos and bugs
- remove C++ overhead ('magic words') and make the code more readable (eg. access to pat containers and declaration of conditions in just one line of code).
- unify the naming of options
- easier 'bookkeeping': standard monitoring histograms and summary boxes get automatically the name of the algorithm
- allow that all the physics functions to be used in cuts are maintained in a central way
- centralized automatic standard content for summary boxes and monitoring

Note that the base class will not remove any flexibility or functionality. It will just add some shortcuts for tasks that are common!

Some more detail on the functionalities that would be included:

a) Physics functions
The base class would contain a collection of physics functions. As an example, IPMin(Track& track) returning the minimum IP to any PV. These functions would be called with such a simple syntax. The code being maintained centrally, it will be possible to optimize the time or memory performance of the functions for the whole HLT.
These functions would also have 'memory', in such a way that the IP of a track that has already been computed will not be recomputed.

b) Input/output
The base class will take care of accessing the PatDataStore and providing the pointers for tracks and vertices. It will support one or two input containers for tracks and one for PVs. If the second input container for tracks is not provided by options, the class will consider that it is not there.

Input tracks would be available in alley algorithms just by asking for m_inputTracks or m_inputTracks2, without any previous ‘magic words’ neither in the .h or the .cpp files. In a similar way, the base class will support two output containers for tracks and vertices.

c) Decision
The “Filter” option would be provided for debugging or developing purposes. If it is set to false, the base class will override the decision set in filterPassed in order for the following algorithms in the sequence to be executed. The base class will count the number of events in which the algorithm decision was overruled.

d) Interaction with conditions data base
All the overhead will disappear from the .h and .cpp files. The method to get a variable from the condition data base will be as simple as declareCondition ("MuIPMin",m_muIPMin). The conditionPath option would be available to specify the path where the conditions are defined in xml.

e) Monitoring and debugging
The base class would take care of monitoring common features: number of objects in input/output containers, number of times that the containers are empty, number of times that the algorithm is executed, number of accepted events, number of overruled decisions. The syntax for the monitoring of variables/histograms which are specific of each algorithm would also be simplified.
In addition, the infrastructure for dealing with different levels and rates of monitoring will also be supplied.

f) Dealing with the summary box
If requested by options, the pointer to the summary will be provided by the base class, as well as methods to incorporate information to the summary box.

2. Common algorithms

We have realized that most (not all!) of the algorithmic part of the Hadron and Muon+Hadron alley can be implemented by a succession of only two algorithms, alternated with reconstruction stages. The usage of these ‘small boxes’ can make the code more modular and the maintenance simpler. Moreover, this would make possible that most of the logics of the alleys is readable from the option files, and that all the common monitoring in section 2.e) is done automatically at every step.

These algorithms are of course built on the functionality of the base class. We encourage the developers of other alleys to try the effect on clarity and time performance of using them where suitable.

The algorithms are:

a) Filter Tracks:
Takes a collection of input tracks and filters them according to a list of cuts that are passed by options (p, pT, IPS, IPSMin, IPPositiveMin…). As output, it produces a logical container of tracks, which then can be passed as input to any other algorithm.
The so called logical container will allow passing a list of objects from one algorithm to another without the necessity of re-creating them physically.

b) RelativeCuts:
Takes two collections of input tracks and filters them according to a list of cuts that are passed by options (angle between them, distance of closest approach, vertex quality…). As output, produces a logical container of vertices or tracks, which then can be passed as input to any other algorithm.

Both algorithms will do setFilterPassed(false) if no tracks pass the cut, producing interruption of the sequencer.

3. Guidelines

a) Cuts

- The default value of the cuts should be such that if the value is not provided by options the cut remains inactive. This will allow an easy understanding of the option file or of the effect of the values set in the condition data base.
- The order of cuts in the routines must be optimized for timing: apply first cuts on quantities that are easy to compute, exit from loops as soon as cuts are not passed, etc.

b) Decision

- All algorithms must make their decision public to Gaudi by an appropriate use of setFilterPassed(). This will make possible the effective use of GaudiSequencer. The decision can be overruled by options by the usage of the Filter option.

- All reconstruction algorithms must set filterPassed to false if no output objects are produced, or if it is number is unreasonable (for instance, if the number of PVs is above a given threshold).

c) Messaging

- Verbosity: debug() messages should contain a summary of the result of the algorithm on an event (a few lines per event). More verbose information (like per track information) should be assigned to verbose(). In algorithms containing several methods, it would convenient that each of them prints with the adequate level of verbosity its result.
- Avoiding CPU overhead: all debug/verbose messages in the execute() routine must be enclosed in loops of the form “if (msgLevel(MSG::VERBOSE)) { }”.