Analysis environments for CMS

To cite this article: C D Jones et al 2008 J. Phys.: Conf. Ser. 119 032027

View the article online for updates and enhancements.

Related content
- Recent Standard Model results from CMS
  Simon de Visscher and CMS collaboration
- Upgrades for the CMS simulation
  D J Lange, M Hildreth, V N Ivanchenko et al.
- Optimizing High-Latency I/O in CMSSW
  Brian Bockelman

Recent citations
- Michele De Gruttola et al
- Francesca Cavallari et al
- Persistent storage of non-event data in the CMS databases
  M De Gruttola et al
Analysis Environments for CMS

C. D. Jones, L. Luca, B. Hegner
Cornell University, Ithaca, NY 14853, USA
E-mail: cdj@mail.lepp.cornell.edu

Abstract. The CMS offline software suite uses a layered approach to provide several different environments suitable for a wide range of analysis styles. At the heart of all the environments is the ROOT-based event data model file format. The simplest environment uses "bare" ROOT to read files directly, without the use of any CMS-specific supporting libraries. This is useful for performing simple checks on a file or plotting simple distributions (such as the momentum distribution of tracks). The second environment supports use of the CMS framework's smart pointers that read data on demand, as well as automatic loading of the libraries holding the object interfaces. This environment fully supports interactive ROOT sessions in either CINT or PyROOT. The third environment combines ROOT's TSelector with the data access API of the full CMS framework, facilitating sharing of code between the ROOT environment and the full framework. The final environment is the full CMS framework that is used for all data production activities as well as full access to all data available on the Grid. By providing a layered approach to analysis environments, physicists can choose the environment that most closely matches their individual work style.

1. Introduction
When it comes to the final stages of an analysis, there are almost as many strategies as there are physicists. Some may prefer to use the experiment’s proprietary data processing framework to run computationally intensive jobs while others may want to do quick plots using an interactive environment. In order to accommodate as many of these strategies as possible, CMS has taken a layered approach to its software design. The first is CMS’ full offline framework which is used for processing on the grid and can be used locally. The second is TFWLiteSelector which can be used in ROOT [1] or with PROOF [2] for long running local or LAN work. The third is FWLite which works in ROOT using full object dictionaries and is typically used for local short-duration work. The fourth is 'bare' ROOT which can be used for simple interactive activities. Migration between these different environments is accomplished via a common data model and ROOT file format.

2. Data Model
The CMS data model is fairly typical for an HEP experiment. The primary data concept is an Event which contains data products. E.g., a list of tracks is contained within the Event. The data products are constructed by a module (the smallest unit of work in the CMS full framework) while running in a process. E.g., The track finder module creates the list of tracks while in the reconstruction process.

Data products are uniquely identified using four pieces of information:
C++ class type Required for type-safety, e.g., `std::vector<Track>`.

module label A unique string which was assigned to the module which constructed the object, e.g., "trackFinder".

product instance label If a module inserts multiple objects of the same type into the event, this string is used to differentiate the different products, e.g., "" or "failFit". If no product instance label is specified during a data access then we use an empty string as the default since that is the same default value used when data products are registered.

process name A unique string assigned to the process being run. This keeps data from different processing steps (e.g., HLT or Reco) from interfering. If no process name is given during a data access we default to the most recently run process which has a match for the other three pieces.

We provide support for inter-product links via a CMS specific smart pointer, `edm::Ref<>`. For example, tracks contain a list of `edm::Ref<>`’s to the hits from which they are formed. Unlike a `TRef` which can only return an item on a different branch if that item has already been retrieved, the `edm::Ref<>` can communicate with an external service to cause data to be read from a different branch.

The primary data format for CMS is a ROOT based format. In this format, the transient and persistent representations of the data products are identical. This allows the data products to be read using auto-generated ROOT dictionaries via the Reflex system. The CMS ROOT file format uses one `TTree` to contain the events with each data product having its own ` TBranch` as direct children of the events `TTree`. The names of the ` TBranches` are formed by concatenating a simplified version of the C++ class name of the class held by the ` TBranch` with the three string labels. Some example branch names are "Tracks_trackFinder_Reco" and "Tracks_trackFinder_failFit_Reco". Instead of directly holding the data product, the ` TBranches` actually hold an `edm::Wrapper<>` class and it is the wrapper which holds the data product. This allows additional status information, e.g., is the data product actually missing from this event, to be stored as well as giving a common base class (`edm::EDProduct`) which provides a standard virtual destructor in order to properly delete objects once the event has been processed. Because CMS uses the auto-generated dictionaries, we can make use of full splitting when storing the data products. The full splitting improves compression on the disk as well as readback performance.

In addition to the `TTree` holding the events, the file format also contains `TTree`s which contain provenance information about how the data products where created. This information includes exactly which data products were read by the module for that event when the data product was created.

3. Full Framework

The CMS full framework is used for all large scale production work: HLT, simulation, prompt reconstruction, re-rereconstruction, alignment, calibration and skimming done by the physics groups. In addition, the full framework is the only supported environment for physicists to use with the grid.

To use the full framework, a physicist declares the name of the process to be run along with which modules should be used in the job. The module declaration determine from where data should be read, how new data should be created and where as well as which data should be saved to the external ROOT file.

Within the full framework, modules are passed an `edm::Event` instance and then data is requested from it. Several different member functions of `edm::Event` may be used to access data. In all cases, the data access is kept type-safe by passing an `edm::Handle<>` instance to a templated member function of `edm::Event` and using the type of the class held by the
edm::Handle<> to generate the key used to find the data. The most used function is one which finds the most recent process which contains a matching C++ type and label combination:

```cpp
//find the most recent process with this label
void MyModule::analyze(const edm::Event& iEvent, ..) {
    edm::Handle<std::vector<Track> > tracks;
    iEvent.getByLabel(trackFinder,tracks);
}
```

In the rare case where the most recent process may not be the one which holds the needed data, all labels may be explicitly specified:

```cpp
//specify all the labels explicitly
void MyModule::analyze(const edm::Event& iEvent,..) {
    edm::Handle<std::vector<Track> > tracks;
    iEvent.getByLabel(trackFinder, , Reco, tracks);
}
```

In both these cases, if the data is not contained within the edm::Event then an invalid edm::Handle<> is returned. If the invalid edm::Handle<> is subsequently dereferenced, a C++ exception is thrown. If an edm::Event is unable to deliver data for any other reason, e.g., attempts to read the requested data from disk fails, then a C++ exception is thrown directly from the templated member function.

If the selection criteria needed to obtain data is complex, a physicist may also use a class inheriting from edm::Selector to specify her criteria. However, since the selection may match zero, one or many products in the edm::Event, this member function fills a std::vector<edm::Handle<> >.

```cpp
//Use a selection functor to find a match
void MyModule::analyze(const edm::Event& iEvent,..) {
    std::vector<edm::Handle<std::vector<Track> > > trackH;
    iEvent.getMany(MySelector(), trackH);
}
```

### 4. TFWLiteSelector

ROOT also supports a mechanism to have a code module implicitly loop over data in order to process that data. To use this mechanism, one must write a C++ class which inherits from TSelector and implements the proper virtual functions. This mechanism is also used by PROOF and therefore is useful for processing medium length analyses (a few to tens of minutes) which can be run locally on one machine or on a PROOF farm on the LAN.

In order to allow physicists to maximize the reuse of their knowledge about the full framework, we have made it possible to use an edm::Event from within a TSelector. This is accomplished by having the physicist inherit from the CMS specific TFWLiteSelector<> class rather than directly from TSelector. In addition to providing access to a edm::Event, the TFWLiteSelector<> modifies the interface presented by a TSelector in order to make it obvious which methods are run on the ROOT client and which are run on the PROOF servers. The client parts are the virtual methods of the TFWLiteSelector<> class itself while the server parts are methods of a Worker where the Worker is declared via the template argument to TFWLiteSelector<>. Only one TFWLiteSelector<> is created per job but a new Worker is created on each server. The client and server parts communicate by explicitly passing TLists from one to the other (which is what happens implicitly in the TSelector). A simple example is shown below

```cpp
//The client side part
class MySelector :public TFWLiteSelector<MyWorker> {
    void begin(TList*& toWorkers) {}
}```
void terminate(TList& out) {outFindObject(a)->Draw();}
};

//The server side part
class MyWorker {
  MyWorker(const TList* fromCLient, TList& out):
    h_a(new TH1F(a,...) )
  {out.Add(h_a); }

  void process(const edm::Event& iEvent) {
    edm::Handle<std::vector<Track> > tracks;
    iEvent.getByLabel(trackFinder,tracks);
    h_a->Fill(tracks.size()); }

  void postProcess(TList&) {}
};

In this example, a physicist would create an instance of MySelector in ROOT. When data processing begins, ROOT calls MySelector::begin passing a TList* which can be assigned to a new TList and filled with objects you wish to pass to the workers, in this case nothing is to be passed. Once MySelector::begin has finished, an instance of MyWorker is created on each server (or one is created in the client if PROOF is not being used). When created, the MyWorker creates its own histogram and keeps a pointer to that histogram as well as adding that histogram to the list of objects which will be returned to the client. Each server loops through its list of events and for each event it calls the MyWorker::process method. In the process method, the code gets the lists of tracks and adds its the size of the list to the histogram. Once processing of the events is completed for a server, the MyWorker::postProcess method is called and the worker has a final chance to add something to the TList which will be sent back to the client. Once all servers have finished processing their events, the TLists from all of the servers are merged together and the final merged TList is passed to the MySelector::terminate method where the merged histogram is drawn.

5. FWLite
Not every physicist finds it productive to work in the constrained environments of the full framework or ROOT’s TSelector. For these physicists, we allow direct calls to ROOT’s I/O system using the class dictionaries and some helper classes. In this way, physicists can (and in fact must) manage the event loop explicitly. We refer to this type of environment as framework lite or FWLite. This environment is useful for interactive analysis running on one machine via the use of ROOT’s CINT interpreter or compiled code. The use of CMS specific helper classes allows this environment to use CMS’ intra-Event smart pointers (the edm::Ref<s>) and provides automatic loading of the proper dynamic libraries via CMS’ plugin system.

This environment supports three different, but related, access mechanisms: dictionaries only, fwlite::Event, and Python.

5.1. Dictionaries only
As the name implies, the dictionaries only mechanism is where only the auto library loading and intra-Event smart pointer aspects are used and the physicists directly gets their data by calling the standard ROOT I/O interfaces. Below is shown data access using the full framework and using the dictionary only method in order to see the differences. First is the framework:
edm::Handle<std::vector<Track> > tracks;
iEvent.getByLabel(trackFinder, tracks);

Second is the dictionaries only:

```cpp
// Load the autoloader
gSystem->Load("libFWCoreFWLite");
AutoLibraryLoader::enable();

TFile f(...);
TTree* ev = (TTree*) f.Get(Events);
ev.GetEntry(); // Needed so SetAddress will work correctly
```

```cpp
std::vector<reco::Track> tracks;
TBranch* bTracks = ev->GetBranch(Tracks_trackFinder__Reco.obj);
bTracks->SetAddress(&tracks);
```

```cpp
for(int index=0; ...) {
  bTracks->GetEntry(index);
  ev->GetEntry(index,0); // Needed for edm::Ref
}
```

The dictionaries only method presents a series of difficulties for the physicist. The first is one must know how to interpret the branch names. One must know which C++ classes go with which branches. One must also know if there are multiple branches which only differ by the last word (which is the process name) then one must know how to find out which one is the latest. In addition, one must know which data member of the `edm::Wrapper<>` class corresponds to the data product.

A second difficulty is the fact that setting the address to the buffer used by the `TBranch` is error prone. It is very easy to pass the wrong object type to the `TBranch::SetAddress` method since there is no C++ type checking done. In addition, we have found that one must call `TTree::GetEntry` before calling `TBranch::SetAddress` for a sub branch else the data is never copied to the proper buffer. And lastly, one must know to pass to `TBranch::SetAddress` a pointer to an object if you are dealing with a sub-branch but pass a pointer to a pointer to the object when the `TBranch` is branch directly held by a `TTree`.

A third difficulty is the fact that getting the data itself can be error prone. One must not forget to call `TBranch::GetEntry` before trying to read the data. Also, one must know how to tell if the data object was not put into the Event at all and instead a dummy object was placed in the `TBranch` in order to keep all branches in the `TTree` balanced.

A fourth and final difficulty is the fact that dealing with the CMS intra-Event smart pointers is difficult in this mechanism because there is no way for the system to know when you have advanced to a new Event. To make the smart pointers work, a physicist must call `TTree::GetEntry` for each event before ever asking for an `edm::Ref<>` (which might happen internally to a member function). This is needed to find out which event index is presently being used. In order to cover all the different use cases (e.g. in a macro or using `TTree::Draw`) the code must do many checks and therefore access to the data referred through an `edm::Ref<>` is not super fast. In fact, the access can fail under certain conditions because we have been unable to find sufficient 'hooks' within ROOT in order to determine exactly the state of the processing.

### 5.2. fwlite::Event

We created a helper class, `fwlite::Event`, in order to alleviate the difficulties found with the dictionaries only method. To illustrate, we will again compare to the full framework access method:
edm::Handle<std::vector<Track> > tracks;
iEvent.getByLabel(trackFinder,tracks);

If we use the fwlite::Event helper the access becomes:

```cpp
//Load the autoloader
gSystem->Load("libFWCoreFWLite");
AutoLibraryLoader::enable();

#include DataFormats/FWLite/interface/Handle.h
TFile f(...);
fwlite::Event ev(&f);
for( ev.toBegin(); !ev.atEnd(); ++ev) {
  fwlite::Handle<std::vector<Track> > tracks;
  tracks.getByLabel(ev,trackFinder);
}
```

Besides the explicit looping through events, the biggest difference between data access in the full framework and using fwlite::Event is in the full framework you call a templated member function of edm::Event to get the data while when using fwlite::Event you call a regular method of the templated class fwlite::Handle<>. This was done because CINT has limited functionality when dealing with templated member functions.

The use of fwlite::Event offers many improvements over using the dictionaries directly. First, physicists do not have to know the meaning of the branch names. Instead they just pass the same information they would when using the full framework. Because of this insulation from the branch names, the fwlite::Event can transparently deal with the case where the simplified class name used in the branch is changed from one software release to a new release. Second, the branch memory handling is done by the system. This allows results to be cached for faster access. In addition, it is not possible to assign the wrong C++ class object to the branch. Third, getting data is safe since the data objects can only be accessed after the call to getByLabel. If a problem occurs during the data access call, an exception is thrown. Fourth, edm::Ref<>'s work under all conditions. This is because the code has control of the data access so the code can safely and quickly setup the edm::Ref<>'s.

5.3. Python

Although CINT has an advantage that its syntax is 'C++ like', it has the disadvantage that it is not a main stream scripting language. Within CMS, Python is being used for a number of different tasks including our workflow management system citeprodagentRef. Given the interest in Python, we developed CMS specific Python modules on top of PyROOT [4] in order to help with analysis. Following the previous examples, the equivalent python code needed to access the track collection is

```python
import PhysicsTools.PythonAnalysis as pa
events = pa.EventTree(...root)
for e in events:
  tracks = e.Tracks_trackFinder__Reco()
```

In this code one first imports the Python module and assign it a new namespace alias of pa in order to save typing later. Then we create an instance of a EventTree by passing it the name of the file to process. Finally we loop over the events and for each event we ask for the tracks which are associated with the TBranch named "Tracks_trackFinder__Reco". Because Python objects are very dynamic, the first time the attribute Tracks_trackFinder__Reco is accessed we are able to check to see if there is a TBranch with that name in the "Events" TTree. If one does exist then we create a buffer of the appropriate type and assign it to that TBranch and assign
a 'function like' Python object holding that buffer to the object's attribute. By assigning the object to the attribute the lookup of the branch by name only has to happen the first time the request is made. The first time the attribute is 'called' (i.e. when the '()' operation is applied to the attribute) for a particular event, the 'GetEntry()' is called for the branch and then the data buffer is returned.

The Python code has nearly the same advantages as the fwlite::Event although the access does require one to know the actual branch name. However, this is offset by the fact that one does not have to know the exact data type of the buffer.

6. Bare ROOT
Because CMS uses ROOT's full split mode when storing data, it is possible to use the TBranch and TBranch::Draw to do simple and quick analyses on the ROOT files without the use of a dictionary.

7. Data Model Challenges
Having the transient and the persistent forms of the data be identical does not come without a cost. We have had to be careful about the internal design of the objects and exactly how the schema for those objects change in order to avoid conflicts with ROOT's internal design.

One example showing the interplay between ROOT's design and the design of stored objects has to do with dealing with objects which use temporary internal caches. Take as an example a three vector class which internally uses a cartesian convention. However, it is discovered that many users are asking for the phi value from the three vector and the calculation is taking a significant amount of time. To solve the problem, a new member variable is added to hold the value of phi. This value is initialized to a non-physical value in the constructor. When phi is requested, if the cached value of phi is the non-physical value then phi is computed and the value cached. Because the value of phi can be calculated from x and y, the member variable holding phi is declared transient to ROOT. Such caching is a standard technique in C++ but it runs into problems when used with ROOT I/O. The problem occurs because ROOT encourages developers to (and internally does) reuse the same data objects from one event to the next but only calls the default constructor for the first event. This means on the first event the cache would be set to the non-physical value. Then the first request for phi would cause it be be calculated for that first event and cached. However, on the next and subsequent events, since the cache is never reset (because the default constructor is not called again) when one asks for phi one will get back the value of phi for the first event and not for the present event. To work around this problem, we developed special cache classes. The cache classes define all their data members as transient to ROOT and we also define a custom TStreamer each class. When the TStreamer is called, we call the ROOT autogenerated streaming code for that class and we reset the value of the cache back to its 'default' value so that for each new event the cache is reset.

Schema evolution is also an additional challenge to the data model. ROOT provides some basic schema evolution ability for fully split objects. For one, ROOT gracefully handles the case where a data member has been added or removed between the version of the software which wrote the ROOT file and the version of the software which is reading that file. ROOT also handles the case where a member data is changed from one primitive type (e.g., int) to a different primitive type (e.g., long). However, additional abilities would be useful. One such ability would be to apply a transform to an old member data. This would allow one to change an internal value which was previously stored in millimeters to be changed to centimeters. A second ability would be to apply a transformation to several old data members. This would allow one to read coordinates stored as x,y,z and change them to r,theta, phi. A third ability would be to change a data member from one class to an equivalent class. This would allow one
to read from a file a data member which was formerly a CLHEP three vector but in the present code is now a ROOT three vector class. CMS has been having discussions with the ROOT team to determine possible ways to handle such cases.

8. Conclusion
CMS has made great effort to be sure that the standard framework file format can also be used directly in ROOT. This allows physicists to run large jobs using the framework, study the resulting skim files and then uses those same skim files back in the framework for additional large scale processing. This format has been in use and refined for the past two years and during that time physicists have been giving feedback. Much of this feedback has been for the different FWLite strategies. In addition, the data challenge this Fall will be producing physics skims only in this format therefore the various physics level objects will be exercised by all of the physics groups.

However, creating a data model which is compatible with our variously supported analysis environments has been very challenging. We believe the challenge is worth the effort since physicists will ultimately 'vote with their feet' and it is only with constant interaction with physicists that we will be able to succeed.

9. Acknowledgements
This work was only possible by the efforts of the CMS Offline and Computing team.
This work was partially supported by National Science Foundation and Department Of Energy.

References