THE NEW CMS EVENT DATA MODEL AND FRAMEWORK

C. D. Jones, Cornell University, Ithaca, NY 14853 USA
M. Paterno, J. Kowalkowski, L. Sexton-Kennedy, W. Tanenbaum, FNAL, Batavia, IL 60510, USA

Abstract

The new CMS Event Data Model and Framework that will be used for the high-level trigger, reconstruction, simulation and analysis is presented. The new framework is centered around the concept of an Event. A data processing job is composed of a series of algorithms (e.g., a track finder or track fitter) that run in a particular order. The algorithms only communicate via data stored in the Event. To facilitate testing, all data items placed in the Event are storable to ROOT/IO using POOL. This allows one to run a partial job (e.g., just track finding) and check the results without having to go through any further processing steps. In addition, the POOL/ROOT files generated by the new framework are directly browseable in ROOT. This allows one to accomplish simple data analyses without any additional tools. More complex studies can be supported in ROOT just by loading the appropriate shared libraries that contain the dictionaries for the stored objects. By taking the time now before data taking has begun to re-engineer the core framework, CMS hopes to provide a clean system that will serve it well for the decades to come.

INTRODUCTION

CMS has begun the process of transitioning to a new data model and data processing framework. This paper describes both the data model and the framework.

The new system is governed by several design goals

- **Provide a clear data model to be used by all software**. The system requires that all data is only accessed through the *Event*.
- **Allow for modular testing of the software**. The system uses a component-based architecture in which data is passed from one component to the next via the Event. In addition, all objects in the Event must be storable so that tests can be run on individual components in isolation.
- **Track provenance of all data**. Data files record what software and what other data are used to produce each datum.
- **Browseable ROOT files from production**. This allows for simple validation to be performed on the results of a job. The consequence is data in an Event must be kept as simple as possible.

EVENT

The Event is a C++ type-safe container of all data related to an HEP Event. Any C++ class can be placed in an Event, there is no requirement on inheritance from a common base class. For example, an 'int' could be placed directly in the Event. This freedom lowers the barrier on development and allows the data objects to be used in other applications. The only requirement is an LCG Reflex dictionary [1] must be generated for the class. This class is used to facilitate storage, to allow objects to be printed to a log, etc. Since the LCG Reflex system can handle any C++ class, this requirement has no effect on data class designs.

Data within the Event are uniquely identified by four quantities: the C++ type and three strings. The three strings are

- **Module label**: the label that was assigned in the configuration file to the module that created the data.
- **Product instance label**: the label assigned to the object from within the module. This label is used to differentiate between multiple instances of the same type created by the same module. This label defaults to the empty string.
- **Job name**: the name assigned in the configuration file to the job that created the data.

Accessing Data

The Event class allows multiple ways to access the data it contains. In all cases, the results of the request will be returned in a smart pointer of type *edm::Handle<>*. If the request was supposed to return only one match and that match could not be found or multiples were found, an exception is thrown. Several of the access methods are discussed below.

```
Handle<TrackVector> trackPtr;
event.getByLabel("tracker",trackPtr);
```

In the above, the code requests the instance of the *TrackVector* that is created by the module with the label "tracker".

```
Handle<SimHitVector> simPtr;
event.getByLabel("detsim","pixel", simPtr);
```

In the above, the code requests the instance of the *SimHitVector* that is created by the module that has the label 'detsim' and has the product instance label "pixel". The product instance label is used to differentiate other *SimHitVector*s also produced by the 'detsim' module.

```
vector<Handle<SimHitVector> > allPtr;
event.getByType( allPtr );
```

The previous code requests all *SimHitVector*s presently available in the event.

```
ParameterSelector<int> coneSel("coneSize", 5);
Handle<JetVector> jetPtr;
event.get( coneSel, jetPtr);
```

The previous code uses a Selector to choose data items of interest. This selector looks at the Parameters for the

modules that create *JetVector*s to determine if they have a parameter named 'coneSize' of type 'int' and value 5.

COMPONENT ARCHITECTURE

The data processing framework uses five different types of dynamically loadable processing components.

- **Source**: provides the initial Event instance to be processed. The source creates the Event, giving it the status information (such as event number) and can add data directly or setup a call-back system to retrieve the data on the first request.
- **OutputModule**: stores the data from the Event. The standard OutputModule uses the Reflex dictionary to write the data to a ROOT file.
- **Producer**: creates new data to be placed in the Event.
- **Filter**: decides if processing should continue for an Event.
- **Analyzer**: studies properties of the Event.

These components only communicate via the Event. Configuration of the components occurs at the start of the job using the ParameterSet system.

ParameterSet System

The ParameterSet system is used to hold the configuration information for all components. A *ParameterSet* allows values to be retrieved in a type-safe manner using a string as the key:

```
double value = pset.getParameter<double>("value");
```

To facilitate more advanced data structures, a *ParameterSet* may hold other *ParameterSet*s or even an *std::vector< ParameterSet >*.

All components are passed a *ParameterSet* as the argument to their constructor. Components are fully configured once they are constructed and cannot be reconfigured during the lifetime of the job.

Producer Code Example

The following is an example of the code that defines a Producer.

```
MyProd::MyProd( const ParameterSet& iPS ):
 m_value(iPS.getParameter<int>("value") ){
  produces<MyData>(); //register what is produced
}

void MyProd::produce(Event& iE ,const EventSetup& )
{
   Handle<OtherData> otherPtr;
   iE.getByLabel("other", otherPtr );

   auto_ptr<MyData> myPtr(
                new MyData(*otherPtr,m_value) );
   iE.put(myPtr);
}
```

In the constructor of *MyProd* we see that it uses one Parameter named 'value' of type *int*. It also registers the fact that it creates a *MyData* object via the call to the templated member function *produces*. The *produce* method of a Producer is called when the framework wants the Producer to create its data. The produce method takes as arguments the *Event* and the *EventSetup*. All Event based information is available in the *Event* while all additional non-Event data (such as calibration values) are accessed via the *EventSetup*.[2] In the example, the code requests an instance of the *OtherData* class from a module with the label 'other'. This instance plus the value obtained from the *ParameterSet* are used to create an instance of the *MyData* class. Once the operation is complete, the instance of the *MyData* class is put into the *Event*. If the produce method were to encounter an exception, all data items put into the *Event* would be removed from the *Event*.

PROCESSING MODEL

An illustration of the processing model for the framework is shown in Figure 1.

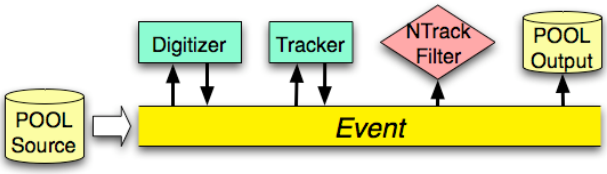

Figure 1: Illustration of the processing model

In the processing model, a Source (in the figure it is the standard source which uses POOL [3]) creates the Event. The Event is then passed to the execution paths. A path is an ordered list of Producer, Filter and Analyzer modules. The paths are expressed explicitly in the configuration file and determine the exact execution order of all the modules. The same module may appear in multiple paths, but the framework will guarantee that a module is only executed once per Event.

When a Producer is called within a path, the Producer reads data from the Event and then, if successful, puts new data into the Event. Filters, on the other hand, read data from the Event and then return a Boolean value that is used to determine if processing of that Event should continue for that path. An Analyzer also reads data from the Event but is neither allowed to add data to the Event nor effect the execution of the path. Finally, when an OutputModule is executed, once all paths have been executed, it reads data from the Event and stores that data to an external media.

If an exception is thrown while processing an Event, the behaviour of the system depends on how the job was configured. The system allows

- skip the module that failed and continue processing the path,
- stop executing the path with the failure and go to the next path, or
- stop execution of the job entirely.

JOB CONFIGURATION

Job configuration is done by parsing a configuration file. The language used by the configuration file was

designed specifically for the CMS framework. An example configuration is shown below.

```
process RECO = {
  source = PoolSource {
             string filename = "test.root"
           }
  module tracker = TrackFinderProducer {}

  module out = PoolOutputModule {
                 string filename = "test2.root"
               }
  path p = {tracker,out}
}
```

The job denoted by this configuration is named "RECO" (denoted by the label that follows the *process* keyword). All data created by this job will be labelled with this job name. In the second line, the *source* keyword states we are defining the source to be used for the job. Following the equal sign is the C++ class of the specific source to be instantiated, in this case the *PoolSource*. Within the curly braces are all the parameters needed to configure the component. The type, name and value are converted into values stored in the ParameterSet that will be passed to the component. On line five, the *module* keyword denotes the specification of a component that can be placed on a path. The string immediately following the module keyword is the module label. The module label is used to specify what path the module should be used within and, if the module is a Producer, it is the module label that is assigned to all data created by the Producer. So in the case of the *TrackFinderProducer*, the list of tracks it creates will be tagged with the module label of 'tracker' and the job name of 'RECO'. The *path* keyword on the second to last line denotes the specification of a particular path. The string following the *path* keyword is the name of the path. The success or failure of a particular path will be record into the output file using the path's name. Within the curly braces are the module labels for the modules that are to be executed in this path in the proper order going from left to right.

PROVENANCE TRACKING

To aid in understanding the full history of an analysis, the framework accumulates provenance for all data stored in the standard ROOT [4] output files. The provenance is recorded in a hierarchical fashion. First, configuration information for each Producer in the job (and all previous jobs contributing data to this job) is stored in the output file. The configuration information includes the ParameterSet used to configure the Producer, the C++ type of the Producer and the software version. Second, each datum stored in the Event is associated with the Producer who creates it. Third, for each Event, the data requested by each Producer when running its algorithm is recorded. In this way the actual interdependencies between data in the Event is captured.

ROOT BROWSING

One main goal of the new EDM is to make the ROOT files read and created by the framework useable directly in ROOT. This would allow physicists to directly use these files. This avoids the HEP standard approach of having to write code for the experiment's proprietary processing framework, which reads the exeriment's standard data format, and then transforms it into a ROOT file more conducive for analysis. Several different ROOT access methods are foreseen: bare, with libraries and framework lite.

In the bare ROOT access, only the output file is used in ROOT, no shared libraries are loaded. The bare access is useful for doing very simple validations of the file and its contents. To make this access method possible, the data within the Event must be kept very simple, for example, C style structs.

In the 'with libraries' ROOT access, the output file and the shared libraries containing the class 'dictionaries' needed by ROOT are used. This gives the physicists full access to all the methods of the classes that were stored in the file. To make it easier to use the 'with libraries' access, we have written a ROOT utility that will automatically load the correct shared library the first time ROOT needs to use the dictionary for the class. This allows physicists to not have to know what libraries contain each C++ class. An example of using the utility is shown below:

```
gSystem->Load("libPhysicsToolsFWLite.so")
AutoLibraryLoader::enable()
TFile f("test.root")
Events.Draw("TrackVector_tracks.size()")
```

In the example, we load the library that holds the *AutoLibraryLoader* class and then enable the loader. Once that is done, we can read in the file, "test.root", and the dictionaries for the classes in the file will automatically be loaded. Once that happens, we can use the member methods of the classes, such as to plot the size of the container that holds the tracks.

The final proposed access pattern, referred to as framework lite, is just in the exploration stage. The plan is to allow physicists to write code in ROOT that looks identical to the code one writes to access Event data in the new CMS framework.

CONCLUSION

The CMS collaboration is completing its transition to a new event data model and data processing framework. This framework draws on experience learned from Babar, CDF, CLEO, D0 and the former CMS framework. The system meets its design goals to

- provide a clear data model,
- allow modular testing,
- track provenance, and
- produce browseable ROOT files.

The first major test of the new system will be the 'magnet test cosmic challenge' that is scheduled to take

place in April of 2006. Once the system is completed, we anticipate that it will serve all of CMS' future needs.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation and the U.S. Department of Energy.

REFERENCES

[1] http://cern.ch/reflex/

[2] C. Jones, "Access to Non-Event Data Access for CMS" International Conference on Computing in High-Energy Physics and Nuclear Physics, Mumbai, India (2006).

[3] http://lcgapp.cern.ch/project/persist/.

[4] .http://root.cern.ch/root/