The New CMS Event Data Model and Framework

Christopher Jones
Cornell University

Jim Kowalkowski
Marc Paterno
Elizabeth Sexton-Kennedy
William Tanenbaum
FNAL

Overview

Design Goals

Event

Component Architecture

Processing Model

Configuration

Provenance Tracking

ROOT Browsing

Design Goals

Provide a clear data model
Data only accessed through the Event

Allow for modular testing
Use a component architecture

Track provenance of data
Data files keep track of how their data was produced

Browseable ROOT files from production
Keep data in Event as simple as possible

Event

Type-safe container for all data related to an HEP Event

Any C++ class can be placed in Event

No requirement on inheritance from common base class

e.g., an 'int' can be used

Event data distinguished by

C++ class type

module label

the label that was assigned to the module that created the data

product instance label

the label assigned to object from within the module

defaults to an empty string

job name

name assigned to the job which created the data

Generate LCG Reflex dictionaries for each C++ class

Used for storage, object dumping to text, etc.

Accessing Event Data

Event class allows multiple ways to access data

```
//Ask by module label and default product label
Handle<TrackVector> trackPtr;
event.getByLabel("tracker", trackPtr );

//Ask by module and product label
Handle<SimHitVector> simPtr;
event.getByLabel("detsim","pixel" ,simPtr );

//Ask by type
vector<Handle<SimHitVector> > allPtr;
event.getByType( allPtr );

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

Component Architecture

Five types of dynamically loadable processing components

Source
Provides the Event to be processed

OutputModule
Stores the data from the Event

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

Components only communicate via the Event

Components are configured at the start of a job using a ParameterSet

ParameterSet

Holds the configuration information for the components

Values are retrieved in a type-safe manner using a string key

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

ParameterSets may hold other ParameterSets

All components are passed a ParameterSet in the constructor

Components are fully configured once constructed

Producer Code Example


```
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);
}
```

Processing Model

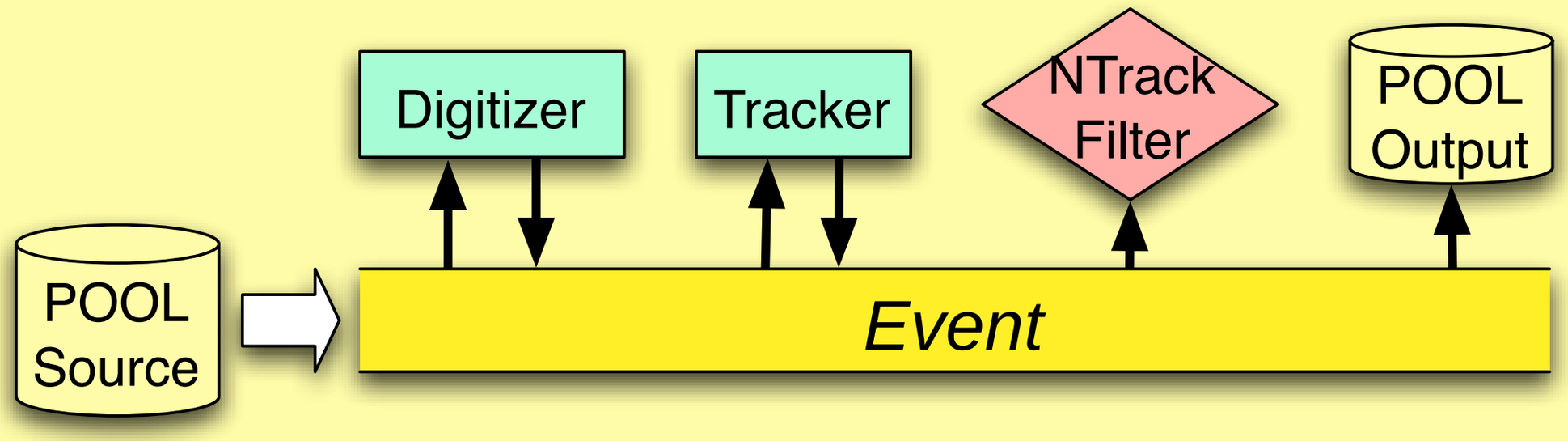
Source creates the Event

The Event is passed to execution paths

Path is an ordered list of Producer/Filter/Analyzer modules

Producers add data to the Event

OutputModule given Event if certain Paths run to completion



Job Configuration

Job configuration is done in the configuration file

```
process RECO = {
   source = PoolSource {
                  string filename = “test.root”
            }
   module tracker = TrackFinderProducer {}
   module out = PoolOutputModule {
                     string filename = “test2.root”
                  }
   path p = {tracker,out}
}
```

After configuration is complete, all components will have been loaded into the application

Provenance Tracking

Configuration of Producers are stored in output file
E.g., Jet Producer labeled 'cone5' used parameter *coneSize = 5*

Data is associated with configuration of Producer
E.g., JetVector with labels ('cone5','','PROD') was made by Jet Producer labeled 'cone5'

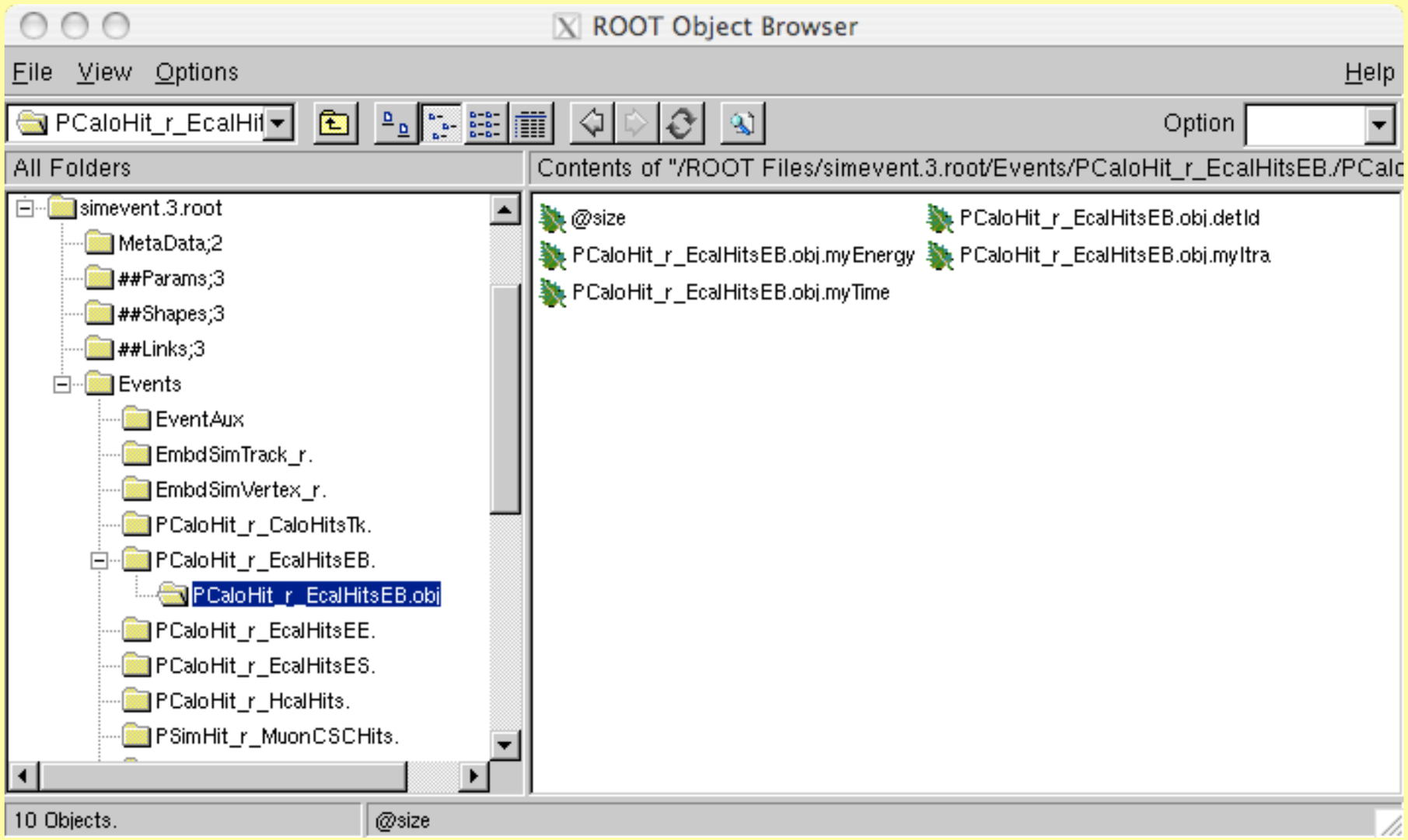
Data requested by Producers are recorded for each Event
E.g., Jet Producer labeled 'cone5' used SuperClusterVector 'clusters' on Event 12

Provenance information stored in ROOT file

Bare ROOT Browsing

Objects in Event are kept as simple as possible

Can do simple tests in ROOT without the libraries



ROOT with Libraries

Able to auto-load libraries when class first needed by ROOT

This gives access to ROOT dictionaries

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


CORNELL UNIVERSITY LEPP
LABORATORY FOR ELEMENTARY-PARTICLE PHYSICS

Conclusion

CMS is completing its transition to a new system
- Uses experience from Babar, CDF, CLEO, D0, and the previous CMS system

System meets its design goals
- Provide a clear data model
- Allow modular testing
- Track provenance
- Produce browseable ROOT files

First major test will be the ‘magnet test cosmic challenge’ in April 2006

Anticipate the new system will serve CMS’ future needs