Table of Contents

2.3 CMSSW Application Framework

Complete:
Detailed Review Status

Contents

Goals of this page

When you finish this page, you should understand:

- the modular architecture of the CMSSW framework
- what comprises an Event
- how data are uniquely identified in an Event
- how Event data are processed
- the basics of provenance tracking

Introduction: CMSSW and Event Data Model (EDM)

The overall collection of software, referred to as CMSSW, is built around a Framework, an Event Data Model (EDM), and Services needed by the simulation, calibration and alignment, and reconstruction modules that process event data so that physicists can perform analysis. The primary goal of the Framework and EDM is to facilitate the development and deployment of reconstruction and analysis software.

The CMSSW event processing model consists of one executable, called cmsRun, and many plug-in modules which are managed by the Framework. All the code needed in the event processing (calibration, reconstruction algorithms, etc.) is contained in the modules. The same executable is used for both detector and Monte Carlo data.

The CMSSW executable, cmsRun, is configured at run time by the user's job-specific configuration file. This file tells cmsRun

- which data to use
- which modules to execute
- which parameter settings to use for each module
- what is the order or the executions of modules, called *path*
- how the events are filtered within each path, and
- how the paths are connected to the output files

Unlike the previous event processing frameworks, cmsRun is extremely lightweight: only the required modules are dynamically loaded at the beginning of the job.

The CMS Event Data Model (EDM) is centered around the concept of an *Event*. An Event is a C++ object container for all RAW and reconstructed data related to a particular collision. During processing, data are passed from one module to the next via the Event, and are accessed only through the Event. All objects in the Event may be individually or collectively stored in ROOT files, and are thus directly browsable in ROOT. This allows tests to be run on individual modules in isolation. Auxiliary information needed to process an Event is called *Event Setup*, and is accessed via the EventSetup.

You will find more infomation on the CMSSW Framework in WorkBookMoreOnCMSSWFramework.

Events in CMSSW

Events as formed in the DAQ and Trigger System (TriDAS)

Physically, an event is the result of a single readout of the detector electronics and the signals that will (in general) have been generated by particles, tracks, energy deposits, present in a number of bunch crossings. The task of the online Trigger and Data Acquisition System (TriDAS) is to select, out of the millions of events recorded in the detector, the most interesting 100 or so per second, and then store them for further analysis. An event has to pass two independent sets of tests, or Trigger Levels, in order to qualify. The tests range from simple and of short duration (Level-1) to sophisticated ones requiring significantly more time to run (High Levels 2 and 3, called HLT). In the end, the HLT system creates RAW data events containing:

- the detector data,
- the level 1 trigger result
- the result of the HLT selections (HLT trigger bits)
- and some of the higher-level objects created during HLT processing.

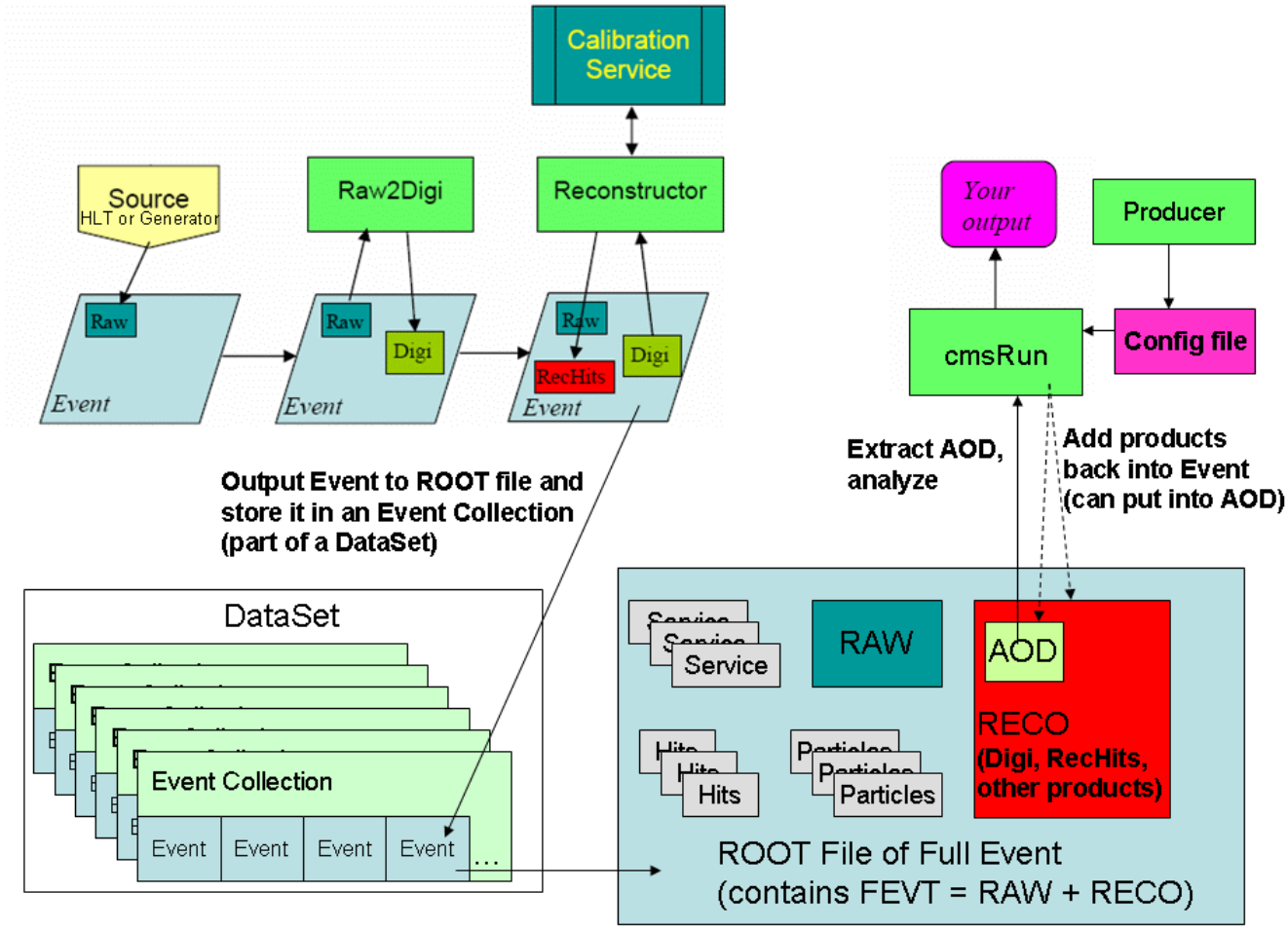
Events from a software point of view: The Event Data Model (EDM)

In software terms, an Event starts as a collection of the RAW data from a detector or MC event, stored as a single entity in memory, a C++ type-safe container called `edm::Event`. Any C++ class can be placed in an Event, there is no requirement on inheritance from a common base class. As the event data is processed, products (of producer modules) are stored in the Event as reconstructed (RECO) data objects. The Event thus holds all data that was taken during a triggered physics event as well as all data derived from the taken data. The Event also contains *metadata* describing the configuration of the software used for the reconstruction of each contained data object and the conditions and calibration data used for such reconstruction. The Event data is output to files browsable by ROOT. The event can be analyzed with ROOT and used as an n-tuple for final analysis.

Products in an Event are stored in separate *containers*, organizational units within an Event used to collect particular types of data separately. There are particle containers (one per particle), hit containers (one per subdetector), and service containers for things like provenance tracking.

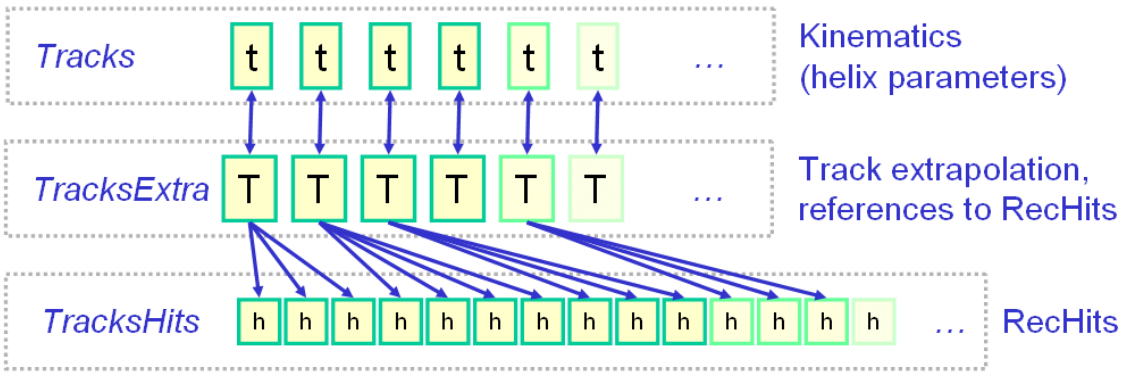
The full event data (FEVT) in an Event is the RAW plus the RECO data. Analysis Object Data (AOD) is a subset of the RECO data in an event; AOD alone is sufficient for most kinds of physics analysis. RAW, AOD and FEVT are described further in the introduction to the analysis. The tier-structured CMS Computing Model governs which portions of the Event data are available at a given tier. For event grouping, the model supports both physicist abstractions, such as *dataset* and *event collection*, as well as physical *packaging*

concepts native to the underlying computing and Grid systems, such as files. This is described in Data Organization. Here is a framework diagram illustrating how an Event changes as data processing occurs:



Modular Event Content

It is important to emphasize that the event data architecture is modular, just as the framework is. Different data layers (using different data formats) can be configured, and a given application can use any layer or layers. The branches (which map one to one with event data objects) can be loaded or dropped on demand by the application. The following diagram illustrates this concept:



Identifying Data in the Event

Data within the Event are uniquely identified by four quantities:

C++ class type of the data
: E.g., edm::PSimHitContainer or reco::TrackCollection.

module label
: the label that was assigned to the module that created the data. E.g., "SimG4Objects" or "TrackProducer".

product instance label
: the label assigned to object from within the module (defaults to an empty string). This is convenient if many of the same type of C++ objects are being put into the edm::Event from within a single module.

process name
: the process name as set in the job that created the data

Modular Architecture of the Framework

A module is a piece (or component) of CMSSW code that can be plugged into the CMSSW executable cmsRun. Each module encapsulates a unit of clearly defined event-processing functionality. Modules are implemented as plug-ins (core libraries and services). They are compiled in fully-bound shared libraries and must be declared to the plug-in manager in order to be registered to the framework. The framework takes care to load the plug-in and instantiate the module when it is requested by the job configuration (sometimes called a "card file"). There is no need to build binary executables for user code!

When preparing an analysis job, the user selects which module(s) to run, and specifies a ParameterSet for each via a configuration file. The module is called for every event according to the `path` statement in the configuration file.

There are six types of modules, whose interface is specified by the framework:

Source
: Reads in an Event from a ROOT file or they can create empty events that later generator filters fill in with content (see WorkBookGeneration), gives the Event status information (such as Event number), and can add data directly or set up a call-back system to retrieve the data on the first request. Examples include the DaqSource which reads in Events from the global DAQ, and the PoolSource which reads Events from a ROOT file.

EDProducer
: CMSSW uses the concept of *producer* modules and *products*, where producer modules (EDProducers) read in data from the Event in one format, produce something from the data, and output the product, in a different format, into the Event. A succession of modules used in an analysis may produce a series of intermediate products, all stored in the Event. An EDProducer example is the `RoadSearchTrackCandidateMaker` module; it creates a `TrackCandidateCollection` which is a collection of found tracks that have not yet had their final fit.

EDFilter
: Reads data from the Event and returns a Boolean value that is used to determine if processing of that Event should continue for that path. An example is `StopAfterNEvents` filter which halts processing of events after the module has processed a set number of events.

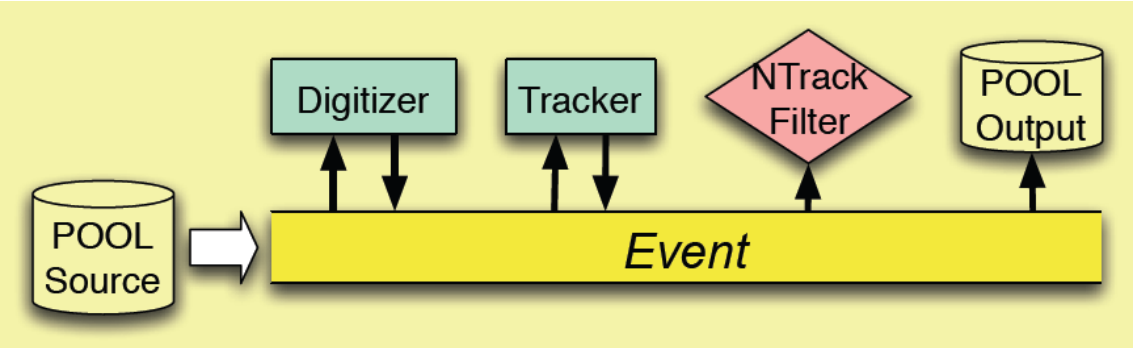
EDAnalyzer
: Studies properties of the Event. An EDAnalyzer reads data from the Event but is neither allowed to add data to the Event nor affect the execution of the path. Typically an EDAnalyzer writes output, e.g., to a ROOT Histogram.

EDLooper

A module which can be used to control 'multi-pass' looping over an input source's data. It can also modify the EventSetup at well defined times. This type of module is used in the track based alignment proceedure.

OutputModule
: Reads data from the Event, and once all paths have been executed, stores the output to external media. An example is `PoolOutputModule` which writes data to a standard CMS format ROOT file.

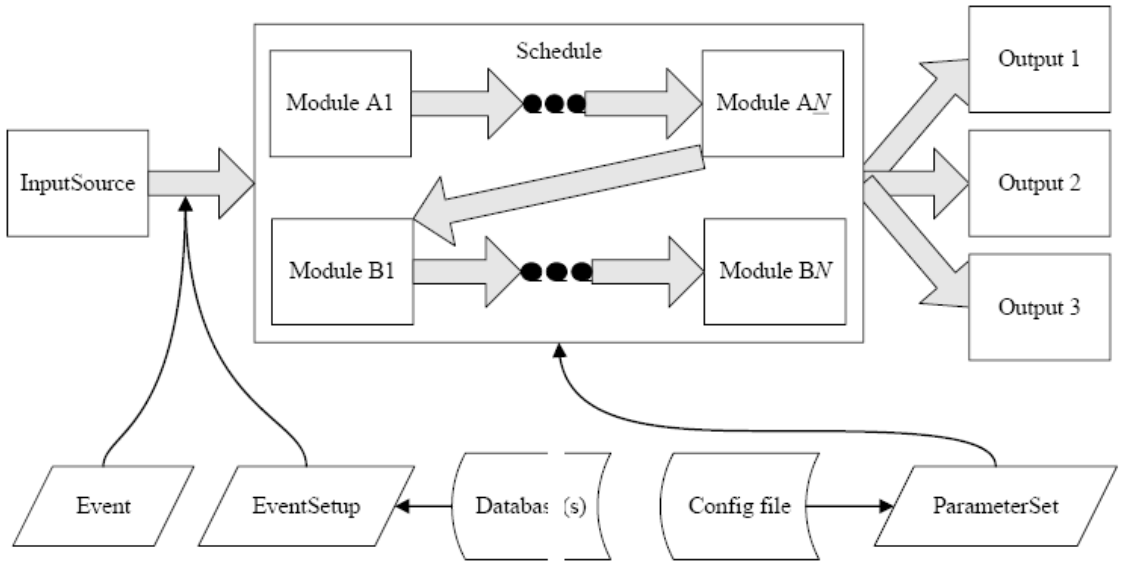
The following sketch shows an example path comprised of a Source ("POOL Source"), two EDProducers ("Digitizer" and "Tracker"), and EDFilter ("!NTrack Filter"), and an OutputModule ("POOL Output"):



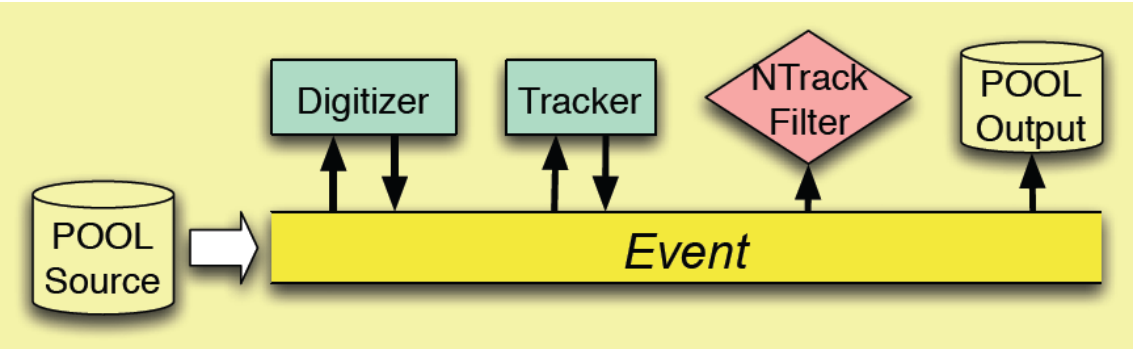
The user configures the modules in the job configuration file using the module-specific ParameterSets. ParameterSets may hold other ParameterSets. Modules cannot be reconfigured during the lifetime of the job.

Once a job is submitted, the Framework takes care of instantiating the modules. Each bit of code described in the config file is dynamically loaded. The process is as follows:

1. First cmsRun reads in the config file and creates a string for each class that needs to be dynamically loaded.
2. It passes this string to the plug-in manager (the program used to manage the plug-in functionality).
3. The plug-in manager consults the string-to-library mapping, and delivers to the framework the libraries that contain the requested C++ classes.
4. The framework loads these libraries.
5. The framework creates a parameter set (PSet) object from the contents of the (loaded) process block in the config file, and hands it to the constructor.
6. The constructor constructs an instance of each module.
7. The executable cmsRun, runs each module in the order specified in the config file.



In a second figure below, we see a Source that provides the Event to the framework. (The standard source which uses POOL is shown; it combines C++ Object streaming technology, such as ROOT I/O, with a transaction-safe relational database store.) The Event is then passed to the execution paths. The paths can then be ordered into a list that makes up the schedule for the process. Note that the same module may appear in multiple paths, but the framework will guarantee that a module is only executed once per Event. Since it will ask for exactly the same products from the event and produce the same result independent of which path it is in, it makes no sense to execute it twice. On the other hand a user designing a trigger path should not have to worry about the full schedule (that could involve 100's of modules). Each path should be executable by itself, in that modules within the path, only ask for things they know have been produced in a previous module in the same path or from the input source. In a perfect world, order of execution of the paths should not matter. However due to the existence of bugs it is always possible that there is an order dependence. Such dependencies should be removed during validation of the job.



-->

Provenance Tracking

To aid in understanding the full history of an analysis, the framework accumulates provenance for all data stored in the standard ROOT output files. The *provenance* is recorded in a hierarchical fashion:

- 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.
- each datum stored in the Event is associated with the Producer which created it.
- for each Event, the data requested by each Producer (when running its algorithm) are recorded. In this way the actual interdependencies between data in the Event are captured.

Access CMSSW Code

The CMSSW code is contained in GitHub CMSSW. You can browse this huge amount of code or search using the CMSSW Software Cross-Reference . You can access it directly on `/cvmfs/cms.cern.ch/slc6_amd64_gcc700/cms/cmssw/`. Packages are organized by functionality. This is a change from the older CMS framework, in which they were organized by subdetector component.

Information Sources

CMS computing TDR 4.1 -- cmsdoc.cern.ch/cms/ppt/tdr; discussion with J Yarba 3/17/06 and with Liz Sexton-Kennedy and Oliver Gutsche.
ECAL CMSSW Tutorial (Paolo Meridiani, INFN Roma, ECAL week April 27, 2006)
Framework tutorial
Physics TDR

Discussion with Luca Lista, and his presentation RECO/AOD Issues for Analysis Tools.

Recovering data from CMS software black hole. G. Zito

Review status

Reviewer/Editor and Date (copy from screen)	Comments
XuanChen - 16 Jul 2014	changed access CMSSW code from cvs to github
PetarMaksimovic - 23 Jul 2009	Drastic simplification to jibe with the new Chapters 3 and 4
Main.Aresh - 19 Feb 2008	change in the index (table tags inserted into TWIKI conditionals for printable version) and in "ExamineOutput" where "PyROOT" arises problems for printable version
JennyWilliams - 23 Oct 2007	review, minor editing
Main.lsexton - 18 Dec 2006	Answered some questions, small changes in phrasing, and corrected the use of the deprecated Event interface found in the G.Zito material
AnneHeavey - 06 and 07 Nov 2006	Reworked "ROOT browsing" section; pulled in info from WorkBookMakeAnalysis; fixed up in prep for mass printing; used G.Zito's EDAnalyzer example

Detailed comments 27-Sep-2012 Hide
I went through chapter 2 section 3. The information is relevant and clear. This section had two links out of date, I updated them.

The entry content " The Processing Model " did not exist any more .

Updated the link at " Framework tutorial "

Responsible: SudhirMalik
Last reviewed by: DavidDagenhart - 25 Feb 2008

This topic: CMSPublic > WorkBookCMSSWFramework
Topic revision: r15 - 2018-07-25 - NitishDhingra