CMS Core Software Re-engineering Roadmap

K. Bloom, W. Brown, V. Innocente, L. Lista, M. Paterno, E. Sexton-Kennedy, W. Tanenbaum, L. Tuura, and A. Yagil
Revision 1.12

Contents

1 Introduction

1.1 Purpose of this Document

This document is a roadmap to aid the re-engineering of the CMS core software: the *event data model* (EDM) and *framework*. It contains many sections that are incomplete, and will continue to do so for the foreseeable future. It contains discussions of requirements, system architectures, design guidelines, details of the design, and plans for the future.

1.2 Structure of this Document

No section of this document is "final". All are subject to change.

Sections marked . . .

Commentary from: Marc Paterno

. . . like this

---

are notes added by one (or more) authors, but which have not yet been "approved" as an official part of the document. Section marked *like this* are specially called out as incomplete sections.

Sections marked with like this:
"Approved on 7 Oct 1571"

have been officially approved. This means that we consider the points in such a section settled. In order to re-open such a point for discussion, one needs to make a persuasive argument that the related analysis is incorrect or incomplete, and to persuade the others that some new analysis is better.

In many parts of this document, we refer to several examples of code, configuration information, and other information. While we strive to make the examples realistic, we note that we are using them for expository purposes only. We do not intend them to be taken a candidate physics algorithms, or trigger algorithms.

1.3 Scope of the Project

The project includes two major sets of deliverables:

1. a software framework for the creation of event-processing applications, (called "the framework") and
2. a software framework for the representation (both in-program and persistent) of collider data, both simulated and real, (called "the event-data model", or EDM).

Included in this project is the required mechanisms to allow event-processing applications to communicate with external systems that perform the other tasks necessary to allow efficient processing of event.

Maybe we should put a list of these different systems, or perhaps instead tasks, here?

1.4 Rationale for the Project

Put a statement of the reason for the re-engineering project here.

2 Requirements

We believe a loose definition of "requirements" is most useful. We have not found it useful to make sharp distinctions between:

1. *constraints*, such as "the code must compile with GCC 3.4.2",
2. *behavioral* or *functional* requirements,
3. *performance requirements*, such as "the high-level trigger must accept x events/s as input, and produce y events/s as output", and
4. *software engineering guidelines*, such as the desire for testability.

2.1 "Physics Requirements" from the CMS Computing Model

The CMS Computing Model [1] specified 34 requirements for the CMS Computing Model. Some of these seem to be of direct relevance to the design of the event-data model and the event-processing framework.

R-1 The online HLT system must create "RAW" data events containing: the detector data, the L1 trigger result, the result of the HLT selections ("HLT trigger bits"), and some of the higher-level objects created during HLT processing.

R-5 Event reconstruction shall generally be performed by a central production team, rather than individual users, in order to make effective use of resources and to provide samples with known provenance and in accordance with CMS priorities.

R-6 CMS production must make use of data provenance tools to record the detailed processing of production datasets and these tools must be useable (and used) by all members of the collaboration to allow them also this detailed provenance tracking.

R-13 The online system will classify RAW events into $\mathcal{O}(50)$ Primary Datasets based solely on the trigger path (L1+HLT); for consistency, the online HLT sofware will run to completion for every selected event.

R-22 A crucial access pattern, particularly at startup will require efficient access to both the RAW and RECO parts of an event.

R-23 The reconstruction program should be fast enough to allow for frequent reprocessing of the data.

R-26 CMS needs to support significant amounts of expert analysis using RAW and RECO data to ensure that the detector and trigger behavior can be correctly understood (including calibrations, alignments, backgrouds, *etc*).

R-27 Physicists will need to perform frequent skims of the Primary Datasets to create sub-samples of selected events.

R-30 Access to information stored in AOD format shall occur through the same interfaces as are used to access the corresponding RECO objects.

R-31 An "Event directory" system will be implemented for CMS.

R-33 Multiple GRID implementations are assumed to be a fact of life. They must be supported in a way that renders the details largely invisible to CMS physicists.

R-34 The GRID implementations should support the movements of jobs and their execution at sites hosting the data, as well as the (less usual) movement of data to a job. Mechanisms should exist for appropriate control of the choices according to CMS policies and resources.

As other requirements from (1) seem to be needed, we can add them here.

Lacking from the list in [1] is a statement of the required *speed* of the high-level trigger.

2.2 Constraints from Software and Computing Management

Explicit processing path scheduling must be supported by this system; the system must support the expression of data-flow paths within job configuration. This feature will exist to support the trigger, where users of the system directly state a path through which an event will pass allowing resources to perform the task to be estimated accurately at program start-up time.

2.3 The CMS Analysis Model

What requirements to we have from the specification of the CMS Analysis Model? What is the official reference that specifies the CMS Analysis Model?

2.4 Requirements from the HLT

Put stuff here regarding how we want specification of high-level triggers to be done as independent paths, so that the designer of a trigger doesn't have to understand other triggers.

It should also describe that we want the trigger program to combine paths to allow for optimal execution.

We also require diagnosis of mis-configured trigger programs at configuration (program start-up) time.

2.5 Grid Computing Support

The event-processing applications need to be able to work in a grid environment. While it is not always clear what this really means, we understand it to mean that an event-processing application must expect to be "packaged" and sent to a target site. The event-processing application must not make such packaging and submission unwieldy.

We do not expect the event-processing application to directly support any particular "grid technology". For example, the event processing application will not itself submit remote jobs, or request event-data files from remote resources.

2.6 Several Typical Use Cases

2.6.1 Case 1: Official Jet Reconstruction with a Cone Algorithm

This is grossly incomplete; we'll add more as we understand what we need.

Task Jet reconstruction.

Goal Use a cone algorithm to reconstruct jets, starting from raw data, putting the reconstruction results into the *Event*.

Actor Physicist who is running the event processing application who wishes to run a specific cone jet algorithm.

Precondition There already exists a file of events containing raw data. The input and output configuration are already given (and are outside the scope of this use case).

Description

1. Make available parameter sets for the tower generating module, the cone jet algorithm, and the jet finder algorithm. We will build calorimeter towers from raw data, relying on an already-built vertex; Then we will build cone jets from these towers, recording full provenance information about the reconstructed information.

2. Express in the task configuration that each of these modules is available

3. Express in the task configuration that the Jets collection production is required (this assumes demand driven will be chosen for this job

4. Identify the process to which this job belongs and the calibration/alignment set needs in the job configuration.

5. generate a configuration bundle featuring all the above data

6. use the identifier for the bundle to submit a job using an approved reconstruction executable

2.6.2 Case 2: Official Jet Reconstruction with a Cone Algorithm

Similar to case 1, but not using official code—rather, using development code, and testing a new algorithm.

2.6.3 Case 3: to be determined

More use cases can go here, if needed. If not, the section title should be fixed.

3 Technology

Put here the technologies we must interact with. For example:

- *POOL and ROOT are to be used for persistency.*
- *PHEDEX is used for . . . what?*
- *. . . lots of other products are used . . .*

4 Architectural Overview

4.1 Responsibilities of Subsystems and External Systems

The task of the event-processing application is to process a sequence of collisions (either simulated or real), with the possibility of producing one or more output files.

Input modules are responsible for obtaining events. Each input module must understand one method of presentation:

- a sequence of event-data files, in the "native format" of the application;
- events presented to the high-level trigger as . . . *Put a description of how the raw data, and the L1 data, are presented to the HLT here.*
- *What other presentation methods do we need? Events provided over a socket, rather than a file? Does this even make sense? Are there others?*

What is the "atom" of event-data for file handling? Are there any rules applied to decide whether two events can be in the same file? What non-event data should be shipped in the file *with the event-data?*

4.2 Major Components of the Infrastructure

The two major components of the core software are:

- the framework for the event-processing application, and
- the event-data model.

4.2.1 Architecture of the Event-Processing Application

We will support two different "styles" of event-processing application in the same software framework. One style of application supports *reconstruction on demand*, in the style of the previous ORCA framework. The other style is more "linear", and is more similar to the style of the CDF and DØ trigger and reconstruction frameworks. We call these styles *unscheduled* and *scheduled*.

4.2.1.1 Commonalities

For both the unscheduled and the scheduled applications, *EDProducer* instances are the objects which actually perform the task of reconstruction. An author of an *EDProducer* does not need to choose to support one or the other style of use; any *EDProducer* is able to be used in either mode.

For both styles of application, the same *EDProduct* classes are used, and the same *EDProduct* instances will be produced from identically-configured *EDProducer*s.

For both styles of application, the same parameter set system is used to configure the *EDProducer*s.

For both styles of application, the same input and output formats are supported.

4.2.1.2 The Unscheduled Application

In the unscheduled application, the action of requesting an *EDProduct* from the *Event* *may* cause the invocation of an *EDProducer*. The high-level view of the mechanism is:

1. User code requests an *EDProduct* through the `Event::get` member template, possibly specifying a selector.
2. The *Event* looks for any already-created objects of the correct type (and which matches the selector, if one was provided). Such objects may be already loaded in memory, or may be retrieved from the input source.
3. If no match was found, the *Event* queries a registry of *EDProducer*s to discover which ones are able to create *EDProduct*s of the correct type (and which could match the provided selector, if any). If no such matches are found, the user will receive an indication that no match is available. No new libraries can be loaded at this time.
4. Any *EDProducer*s found in step 3 are invoked, creating their products and entering them into the *Event*, and possibly causing a cascade of other reconstruction.
5. Any *EDProduct*s generated from the *EDProducer*s just invoked are returned to the user. If no appropriate producers were found, no products may be returned.

An unscheduled application is configured by specifying:

- a selection of independent top-level *EDProduct*s to be written out, or
- a selection of independent high-level triggers to be run, or
- an analysis module to be run, or
- some combination of the above.

and also

- the menu of *EDProducer*s that should be known to the registry of *EDProducer*s.

The combination of *EDProduct*s in the input source and *EDProducer*s registered in the program are the only things that limit the variety of *EDProduct*s than can be obtained from any *Event*.

What other useful ways are there to invoke an unscheduled application?

4.2.1.3 The Scheduled Application

A scheduled application is configured by specifying a module instance path through which the event will flow. More derived or calculated products will be added to the event as it moves through the path.

The responsibility of getting the proper dependency ordering within an explicitly specified path lies with person configuring the job.

4.2.2 Architecture of the Event-Data Model

5 Analysis

5.1 There is more than one source of data

Modules in an event processing application obtain different types of data from different sources. Conditions data come from services. Geometry data come from services. Event data, and data related to collections of events (such as *runs* or *luminosity blocks*) are passed to the modules during the event processing loop.

5.2 Lifetime Management of *EDProducts*

It is important that the lifetimes of the *EDProduct*s be controlled in a deterministic fashion, to avoid resource leaks.

Because the persistent format of the CMS data is based on ROOT, we considered having the *EDProduct* instances allocated directly in ROOT buffers. While this has the beneficial feature of avoiding a memory-to-memory copy of the *EDProduct*, it has several drawbacks that made us decide to *not* choose this design. The drawbacks we identified are the following.

1. It makes a stronger coupling between the *EventPrincipal* and ROOT.
2. This couping makes it much harder to create *EDProduct* instances that will *not* be managed by ROOT. This may be important for the high level trigger, which will process many events that are rejected. Creating and destroying objects in ROOT buffers, and the detailed management necessary to avoid corruption of the files created, may waste critical time in the trigger.
3. This coupling would make it much harder to write the same event-data to several different outputs, whether those outputs are multiple ROOT files or output formats *other* than ROOT.
4. This coupling may make the writing of selected (rather than *all*) *EDProduct*s in an event to persistent storage.

For these reasons, we believe that the cost of a memory-to-memory copy of those *EDProduct*s selected for output is less than the cost of the design that avoids that memory-to-memory copy.

5.3 Unambiguous identification of reconstruction results

It is critical for users to be able to unambiguously identify how each reconstruction result was produced. There are several varieties of information that constitute this identification.

Collectively, we refer to all this information as the *provenance* of the *EDProduct*. Each *EDProduct* is associated with a *Provenance* object that records this information. Where appropriate, *Provenance* objects are shared between *EDProduct* instances.

1. Module configuration

   a) The unique identifier representing all (the names and values) of the run-time configuration parameters given to the module.

   b) A string giving the fully-qualified class name of the module.

2. Parentage

   A vector of the unique identifiers of the *EDProduct*s used as inputs for this bit of reconstruction.

   A module can make use of more than one input to create its output. Should we attempt to specify the types of the EDProduct to which each of the entries in this vector refer?

   Should we perhaps allow a mapping of class name to EDProduct_id?

3. Executable configuration

   a) A string giving the "human friendly" *instance name* of the module which created the product.

      The instance name is unique within the executable—even if there are different instances of the same class, identically configured, in different paths.

   b) A single version number that defines the code for the entire executable. The user can obtain specific library version numbers by querying a central database, using this version number.

      The value is only meaningful for tagged releases.

      This number specifies which libraries were *available* when building the application; it does not indicate that *all* such libraries were used.

4. Conditions Data

   An identifier representing the calibration and alignment set that was used in the construction of this *EDProduct*.

   We assume here that calibration and alignment are handled in the same way and that this single, high-level identifier refers to all the calibration information used for this event. It is possible that individual calibrations (*e.g.*, silicon, calorimeter, muon) will also have IDs associated with them and that each of these will need to be recorded instead of the "set" ID.

   Other conditions data IDs may also be needed here, such as geometry version or hardware configuration.

5. Job configuration

   A physical process name. A job starts up in a particular context such as HLT or Reconstruction. This name identifies the process under which the job was started and is likely to be a run-time property.

All of this provenance data is distinguished from the event data because its principal home is in an ancillary database, although a copy may be readily accessible from the event data (*e.g.*, within the file that contains events).

5.4 Communication between event-processing elements

Clearly the event-processing elements (called here *Modules*) need to communicate—the hits produced by one module will be used to form tracks by another. In order to provide for modular testing, to allow for . . . we require that modules communicate *only* through the event—by putting *EDProducts* into the *Event*. Furthermore, in order to . . . we require that one may "cut" the event-processing chain between any two modules, and save the state of the event at that instant. This requires that all *EDProducts* be *persistable*.

While each *EDProduct* must be persistable, this does not imply each one must be persisted for every event. The event output mechanism must be capable of selective writing of *EDProduct* instances, perhaps to several output streams.

5.5 Input is Not Like Output

There is a lack of symmetry between input and output of events.

In the context of several paths of execution, it is possible to schedule output perhaps to multiple streams in each path of execution. For example, a single event processing executable might contain a path performing W mass analysis, and a second path performing $t\bar{t}$ mass analysis. Each of these paths could usefully write those events interesting to the analysis to its own stream.

Input of events does not have such a similar use. Each job has a single "driving source" of events. This source might read several files, perhaps in parallel. The input still appears to the event processing application as a single stream of events.

For these reasons, we see the need for an *input service*, which is not a module, and for *output modules*.

In general, the framework will invoke the appropriate input service. As a special case, a *mixing module* could invoke an additional input service or services.

This list is very incomplete. Please add additional items you think are needed.

5.6 Schedule Specifications for the HLT

This section describes rules regarding module scheduling and configuration within the HLT.

5.6.1 Concept Definitions

Reconstructor A module whose primary purpose is to perform some step of reconstruction and to place the result into the *Event*.

Filter A module whose primary purpose is to render a decision on the quality of an *Event*, based on the *EDProducts* in the *Event*.

Path The user's expression of requirements regarding which modules (including their configurations) make up a single high-level trigger.

5.6.2 Facts about Concepts

5.6.3 Example Problem

The English description of the problem should go into the use cases section.

We consider the problem setting up a HLT program using two triggers:

1. a top–muon trigger (t_μ), which looks for a high-p_T muon and several jets, and
2. a top–electron trigger (t_e), which looks for a high-p_T electron and several jets.

The t_μ trigger uses jets from the midpoint cone algorithm, and the t_e trigger uses jets from the k_T algorithm.

5.6.3.1 Top (muon) trigger: t_μ

This trigger requires tracks from a specific algorithm (module B_1), which in turn requires unpacking the tracking raw data (module A). It also requires jets from the midpoint cone algorithm (module D), which in turn requires unpacking calorimeter data (module C). Finally, it requires muons (module F), made using tracks from B_1.

5.6.3.2 Top (electron) trigger: t_e

This trigger requires tracks from a specific algorithm (module B_2), which in turn requires unpacking the tracking raw data (module A). It also requires jets from the k_T algorithm (module E), which in turn requires unpacking calorimeter data (module C). Finally, it requires electrons (module G), made using tracks from B_2 and the calorimeter data from C.

5.6.4 An Insufficient Solution

A simple solution to this problem would be to have the user specify each independent trigger by specifying the sequence of modules to be run, in the order in which they are to be run:

- $A \to B_1 \to C \to D \to F$ for t_μ
- $A \to B_2 \to C \to E \to G$ for t_e

This solution is inadequate because it does not convey important information that the user knows and that could be used to compute an optimal schedule. In the case above, this information is that the calorimeter-based modules (C, D and E) are independent of the tracking-based modules (A, B_1, B_2 and F). Because they are independent, the ordering of the calorimeter-based software and the tracking-based software implied by the sequences above are not essential. Only the ordering *within* the tracking set, or the calorimeter set, is essential.

5.6.5 A Sufficient Solution

The critical observation is that each trigger description (such as those above) can be composed from strictly linear sequences which are independent of each other, and which can be combined to produce the full trigger specification.

We see the need for a configuration language richer than the simple one above, which allows specification of:

- (optionally named) sequences of configured modules, which the schedule builder may *not* re-order, (because of implied dependence of later modules on the output of the earlier modules), and
- combinations of (optionally named) independent sequences, with no implication regarding the relative order of the constituent sequences.

Furthermore, we want the elements of a sequence to be either individual modules, or the combinations of independent sequences.

Finally, the schedule builder must be able to read a configuration of the trigger program written in this language, and to perform "optimizations" which do not change the meaning of the program, but which avoid redundant execution of any module.

We can capture the essential features of this configuration language via the grammar in Figure 1. The sequence operator "," is used to express dependencies between modules. It has higher precendence than the & operator, which is used to combine the results of independent sequences. `'Decision'` and `'ConfiguredModule'` are terminal symbols in this grammar.

```
TriggerTerm    ::= PathExpression : 'Decision'
                 | : 'Decision'

PathExpression ::= PathExpression & Sequence
                 | Sequence

Sequence       ::= Sequence , ProcessingUnit
                 | ProcessingUnit

ProcessingUnit ::= 'ConfiguredModule'
                 | ( PathExpression )
```

Figure 1: The configuration grammar for scheduled event-processing applications.

5.7 Event Queries

5.7.1 Definitions

Event The *Event* is an interface through which one gains access to detector output and derived quantities that are associated with a *single collision*.

Selector A *Selector* is a predicate used to identify interesting *EDProduct*s. It encapsulates the user's requirements (*e.g.*, features of interest) for products.

Provenance A *Provenance* carries a snapshot of the data relevant to describing how an *EDProduct* was built. It includes (but is not limited to):

1. Description of the configuration of the module that made the *EDProduct*.

2. Description of the program configuration (*e.g.*, code version).

3. Parentage of the *EDProduct* (*i.e.*, what other *EDProduct*s were used as "inputs" by the *EDProducer* that made the *EDProduct*).

Note that some, but not all, of this information is available at program configuration time.

Getter A *Getter* provides an interface through which a *single EDProduct* is obtained.

Commentary from: Marc and Jim

We think the rest of this section should be removed.

There is more than one variety of event query.

- In *EDProducers*, the query functions can return only exactly one product. If the requested product can not be returned, and exception shall be thrown.
- In an analysis module, an addition query interface is available. These queries may return multiple matching products.

  We have not decided what happens if a failure during reconstruction occurs. How does the data indicate that an algorithm was tried, but failed to complete? Perhaps we can make the ROOT branch entry contain a 2-tuple, consisting of the constructed object (if any) and an error report object (if any). Only one of the two items would actually appear in any entry.

6 Design of the Core Infrastructure

6.1 The *Event*

There will be only one *Event* class.

Purpose: Responsible for managing lifetimes for each *EDProduct* it contains. Manages relationships between *EDProduct* and metadata. Provides access to event data (*EDProducts*) for any consumer of event data. Allows communication between “modules.”

A single *Event* instance corresponds to the detector output, reconstruction products, and/or analysis objects from a single crossing or the simulation of a single crossing.

Event is a concrete class.

It is possible to allow different *Event* interfaces, or merely different member functions, some of which perform ROD, and some of which do not.

- Any *EDProduct* should be immutable after insertion into the *Event* (see §6.2.2).
- The *ParameterSet* provenance of input objects to a particular *EDProduct* should survive the dropping (*dropping* means not writing to the output file) of the original input object.

The *Event* will use methods of the *Selector* class (see §6.4) to search for *EDProducts* matching a given criterion.

An ancillary class of the *Event* will keep track of the full invocation sequence

1. `EDProducer::produce`,
2. `Event::make`,
3. `Event::get`.

This information will be used to build a provenance "record" to be associated with the *EDProduct*.

6.2 *EDProducts*

Purpose: The basic unit of event data managed by the *Event*.

EDProduct is an abstract base class. Derived classes are also referred to as *EDProducts*. Each instance of such a class represents a component of an event, and is capable of persistence.

Each *EDProduct* instance has an ID that is unique within the event.

A "map" of the *EDProduct* instances for an event is kept in the event store.

Commentary from: Luca Lista

Using a generic approach may shield the end user from exposure to the *EDProduct* class, allowing the use of any type, not only *EDProduct* subclasses. This is done, for instance, in BaBar using the *ProxyDict* technique [2].

For "bare root" access using native types (or an STL container of native types) instead of specialized types could also be an advantage.

This is actually already implemented in Marc's prototype, where the class template *EDProduct*$<T>$ inherits from *EDProductBase*.

Commentary from: Lassi Tuura

My understanding was that this was a design choice, not a technical limitation (i.e., users should be aware of EDProduct, and nothing else but EDProduct is allowed into the store).

After all we started out from a whole stack of proxies.

An *EDProduct* that needs to be readable by bare ROOT may contain only built-in data types (*e.g.*, float, double, int), and must have the same shape in its transient and persistent forms. The data members of such a class should have meaningful names and allow simple use. Those *EDProducts* that need not be readable by bare ROOT (*e.g.*, raw data) may be packed and may or may not require additional software in order to be unpacked for browsing.

Each class that represents an *EDProduct* should be as simple as is feasible (with respect to the four usage patterns we have documented). In particular, usage pattern 4 objects (*i.e.*, objects that need external data to be usable) should be used only when necessary (for functionality or performance).

*EDProduct*s are often collections, but they are not required to be. They should not be small.

6.2.1 Common Bookkeeping Information

There are several purposes for saving bookkeeping information:

1. To allow users to identify the *EDProduct* they want by identifying

   a) the type of the *EDProduct*

   b) the name of the “module” *instance* that created it—this is not merely the name of the *class* of that module; it is a name, unique within that executable, that identifies a particular “module” instance

   c) the configuration of the “module” that created it

   d) the calibration data used by the “module” that created it

   e) the processing step that created it.

   f) the release of the software that created it.

   This may not be an exhaustive list.

2. To provide summary information that the user can take elsewhere to look at the actual parameter sets/calibrations/*etc*.

Sufficient bookkeeping information should be stored to allow re-production of the same *EDProduct* instance. This is not yet resolved for *simulation* products; it may be sufficient to reconstruct the entire event. There may also be a problem involving *regional reconstruction*; this seems resolvable by identifying as part of the algorithm the description of the region on which it acted.

This bookkeeping information will be used by the *Selector* class (see §6.4). Some selectors will use *all* the information to make “perfect matches.” Other selectors can use *some* of the information, and then possibly match more than one *EDProduct*.

Each *EDProduct* instance must be associated with its bookkeeping information.

6.2.2 Rules for *EDProduct*-derived Classes

- An *EDProduct* instance should not depend upon the classes that create it.
- An *EDProduct* instance should be immutable once it is it is made persistent.

Despite the immutability of an *EDProduct*, there are two ways in which an *EDProduct* in the *Event* may be augmented:

- extensible collections: in which new objects may be added to collections already in the *Event*.

Commentary from: Brown/Kowalkowski/Paterno

We think that extensible collections are not needed. The functional equivalent can be provided by allowing "view" objects, which can carry information about objects in another collection, and which support (external) iteration over the full set of objects presented in the view.

- decoratable objects in collections: in which a new *EDProduct* is added to the *Event* and is associated with with an *EDProduct* already in the *Event*.

In addition, both "puffing" and "refitting" will be supported.

Puffing means expanding existing data in an *EDProduct*, using no event information from outside that *EDProduct*. Outside *non-event* information (*e.g.*, detector geometry) used in creating the original *EDProduct* may be reused.

Refitting means generating a *new EDProduct* from an older one, using new and different information from *outside* the original *EDProduct*.

6.3 Modules

The purpose of a module is to encapsulate a unit of clearly defined event-processing functionality, in an independently testable and reusable package.

6.3.1 General Characteristics

Here are some characteristics of *Modules*:

Modules is the generic term for all "workers" in the framework. Not all modules have the same interface.

Modules are scheduled by the *ScheduleBuilder*, and invoked by the *ScheduleExecuter*. Each *Module* instance is configured with a *ParameterSet*.

Modules must not interact directly with (*i.e.*, call) other modules.

Only *Modules* are "configurable." An internal algorithm is configured by "percolating" *ParameterSets* to the algorithm, by the *Module* that contains the algorithm.

6.3.2 Types of Framework Modules

Here is a (possibly non-exhaustive) list of framework module types:

- event data producers—reconstruction modules
- mixing
- output

- filter
- analyzers (read-only)

Note that *input* provided by a service, not by a module—see 6.3.5.

6.3.3 *EDProducers*

The only service of an *EDProducer* is to produce *EDProduct* instances and placing them in an *Event*. This service is performed by its `produce(Event& ev)` method.

On invocation a transaction is started.

The *EDProducer* will create empty *EDProducts* by asking the *Event* to make them

```
Handle<EDP> it = ev.make<EDP>();
```

At this point, the *EDProducer* is ready to populate this *EDProduct* with the real reconstructed objects.

If its algorithm requires information from the event, it will get it from the event-store using its `get(vector<Handle<EDP2> >& edps, const Selector& s)`.

The following interface is tentative. We have not resolved all the issues relating to the responsbilities of the various components. Return types are not yet indicated.

Interface of a *EDProducer*.

```
??? beginRun(RunRecord&);
??? beginLumSec(LumSecRecord&);
??? processEvent(Event&);
??? endLumSec(LumSecRecord&);
??? endRun(RunRecord&);
```

We also expect:

- The *Event* will provide access to the *LumSecRecord* to which it belongs, and to the *RunRecord* to which it belongs.
- the *LumSecRecord* will provide access to the *RunRecord* to which it belongs.

These functions all correspond to “run state transitions”.

There may also be other sorts of transitions, not corresponding to run state transitions:

- beginning and ending of a *file*,
- beginning and ending of a *job*,
- *other to be discovered.*

These transitions are *program state* transitions.

6.3.4 Mixing Modules

A *MixingModule* takes in a sequence of `const` *Event*s and merges corresponding data objects from each into a single output merged *Event* which is passed back to the framework. This is its only purpose.

6.3.5 Input and Output Modules

InputModule is an abstract base class.

The *InputModule* class provides the “interface” to read objects from the “I/O system.” A “Database” model will be used, that is, specific *EDProduct* instances will be explicitly retrieved.

We discussed how the *InputModule* uses the data management system to deliver requested events to the “user,” who specifies things like a “process step,” “code version,” *etc*. The data management system resolves this to a set of files, but that isn’t enough—because the user wants only some of the events in those files. The data management system could also deliver an “event catalog” that says what events are to be included. We have agreed that an event catalog is important.

CDF notes that a system that requires strict file delivery order causes trouble. Such an ordering can avoid thrashing on “conditions data.” But the cost has been large for CDF. Creation of an event directory reduces the need for strict file delivery ordering.

Event directories can live either in the data files (such as an AOD) or in their own files. Different event directories can refer to the same data files. It seems critical that a given process use whatever event directory the user “points at.”

6.4 *Selectors*

Selectors provide the mechanism by which one specifies what pieces (*EDProduct*s) of an event are of interest. They are the “query mechanism” of the EDM.

The *Event* uses *get* methods of the *Selector* class to search for *EDProduct*s matching a given criterion. Internally, the *get* methods use the bookkeeping information to determine which *EDProduct*s are a match.

In its main *get* method, `match(const Handle<EDP>& edp)`, the *Selector* will search in the event store for *all EDProduct* instances matching *Selector*.

Event also supports a `get(Handle<EDP>& edp, const Selector& s)` method that will produce an error unless there is one and only one *EDProduct* instance matching `s`.

If *explicit scheduling* is being used, the *get* methods only search the existing event store. If *explicit scheduling* is not being used, the *get* methods will find each matching *EDProduct* whether or not it is already in the event store, invoking the appropriate *EDProducer*s as needed.

Commentary from: Luca Lista

6.4.1 Different selection of event products

The most general selection should provide as result more handles to selected *EDProduct*s. A possible interface could be:

```
get( std::vector<Handle<EDP>& handles, const Selector& d )
```

Nonetheless, it may be frequent to request for a single product using a named selection, that could be called *AliasSelector* :

```
AliasSelector sel( "GoldenElectrons");
Event & ev;
Hangle<EDP> & ele;
ev.get( ele, sel );
```

The *AliasSelector* object should be instantiated only once at the initialization of the framework module that hosts the above code

As alternative interface, the above query could be performed using directly a character string:

```
ev.get( ele, "GoldenElectrons");
```

This could introduce a possible performance reduction due to the search by string match. On the other hand, it is likely that the object of the class *AliasSelector* should become a configurable object, whose actual value has to be set via configuration scripts. In that case, the instantiation a at the initialization of the module would be mandatory, and this would make the interface `ev.get( ele, sel );`, with no string search, more natural.

6.5 The Scheduler System

The scheduler system is the subsystem in the framework that is responsible for executing the sequence of reconstruction steps in the appropriate order.

We will use a system that supports two mutually exclusive types of scheduling.

- explicit scheduling
- no scheduling

Which form of scheduling is used is at the option of the user running the program.

The *ScheduleBuilder* is responsible for organizing the network of modules to be invoked, and assuring that they are invoked in the correct order. It builds the schedule used by the *ScheduleExecuter*.

Both *ScheduleBuilder* and *ScheduleExecuter* are concrete classes.

The *ScheduleBuilder* is configured by the same system as the *EDProducer*s.

The *ScheduleBuilder* must know the sequence of *EDProducer*s for each “path,” and how each *EDProducer* is configured.

The *ScheduleExecuter* must assume that each *EDProducer* may request stopping of execution of that "path."

The *ScheduleExecuter* deals with "framework tasks," which may include checking memory usage between *EDProducer* invocations.

The *ScheduleExecuter* should be able to decide what action should be taken upon each return status of a *Filter*.

6.6 The *ParameterSet* System

6.6.1 *ParameterSets*

Some of the elements in a framework application can be configured at run-time by the user. All such elements will be configured by a common *parameter set system*.

A *ParameterSet* contains a collection of name/value pairs, and provides type-safe access to them. The contents of a *ParameterSet* are uniquely identified by a *ParameterSet_id*. The contained values can be anything from the following list:

- bool
- long
- std::vector<long>
- double
- std::vector<double>
- std::string
- std::vector<std::string>
- ParameterSet
- std::vector<ParameterSet>

It is important to note that parameter sets can be nested.

ParameterSets used for *official production* must be registered in a central database. IDs for such parameter sets must be distinguishable from IDs associated with parameter sets not registered in the central database.

ParameterSets can also be *local*; they then are associated with an ID unique *within the data file*. Local *ParameterSets* are stored in the same file as the *Events* with which they are associated.

An entire executable should be configured using a single *ParameterSet*, which contains the many *ParameterSets* used to configure the *Modules* within that executable. Each module should be configured with a single *ParameterSet*.

There should also be a related system of untracked parameter sets. These are similar to *ParameterSets* in how they are presented to the user, but they do *not* have associated IDs, and are *not* tracked in any repository. They are to be used to carry information which does *not* need to be tracked in the bookkeeping system. One example of such information is the verbosity of the logging level used when running a program. Untracked parameter sets should *not* be used to provide any configuration information that affects the physics of reconstruction results.

6.6.2 Identifying Parameter Sets

There will be a central authority to assign unique IDs to *ParameterSet*s and to store those *ParameterSet*s used in official processing. There will be, in addition, a local repository of *ParameterSet*s, in the event data files themselves. This is needed, in part, to allow use of reconstruction code without contacting the global authority—for purposes *other* than official event processing.

*ParameterSet_id*s are calculated from the contents of the *ParameterSet* by the MD5 algorithm, giving a 16-byte identifier. This means if two IDs are different, the parameter sets to which they refer are surely different. If two *ParameterSet_id*s are the same, then it is very likely, but not 100% certain, that the *ParameterSet*s to which they refer are the same.

6.6.3 User Creation of Parameter Sets

A set of tools (such as a GUI parameter set editor) will be provided. Such tools are needed to make creation and manipulation of *ParameterSet*s simple.

6.7 Non-Event Data

To be filled in later.

6.8 Data Management

Commentary from: Luca Lista

The need for input and output modules is specified in section 6.3.5. The main applications will use POOL data format to write and retreive data. It would be convenient to allow multiple input and output modules to run concurrently in the same job; multiple input module, together with an appropriate event mixing module may provide the ability to mix simulated event with real minimum-bias background; multiple output modules allow to write to multiple streams or skims, each with a configurable selection of events and *EDProduct* to be stored, within the same job.

It could be convenient to encapsulate POOL service as well as input and output tasks in specific classes. Namely, we could have the classes *PoolService* to handle common services, like file catalog and creation of caches (*pool::IDataSvc*); *PoolInput* and *PoolOutput* to read and write, respectively to POOL store.

PoolInput and *PoolOutput* require an access to the event product that may be different from the one provided by the *Event* interface. In particular, the user access has to be type-explicit, because the base class *EDProductBase*[1] has to be hidden to the user. *PoolInput* and *PoolOutput* could well use *EDProductBase* polimorphically, without the unneeded complication to "know" the product types, that is unneeded when managing data persistency. For this reason, it could be useful to specify a class *Store* which is used internally by the class *Event*, that provided polimorphic access to *EDProductBase*. This class should be accessible to *PoolInput* and *PoolOutput* with an interface that may be as simple as:

```
bool PoolInput::read( Store & );
```

```
void PoolInput::write( const Store & );
```

`PoolInput::read( Store & )` returns true or false if the event has been read successfully or not (end of event collection reached).

Writing at the same time to multiple files requires some implementation subtleties with POOL references. In particular, if no cross references are present among objects it could be convienent and efficient to use `markMultiwrite` on references to *EDProduct* selected to be stored; in presence of objects cross-references, in the cases where those could be required, `markMultiwrite` does not guaranetee to preserve the correct reference in multiple files, and the most convenient solution could be to use multiple caches.

To be completed

7 Design of Interfaces to Other Systems

8 Development Approach

To be filled in later.

9 Release Management and Testing

To be filled in later.

10 Deployment

Can we refer to some official CMS document here?

---

[1] I noticed that the document doesn't contain (yet) the architecture of *EDProductBase* and the templated subclass *EDProduct*. This should be included in order to define *EDProductBase* in this context.

A Glossary of Terms

It seems useful to agree up a set of terms to use for the various ideas we have been discussing. Here is a working list of the terms we have used. This list is an uneven mixture of items, some of which are very general and some of which are very specific.

EDProduct Abstract base class of "things" stored in the *Event*.

Sometimes we use the term *EDProduct* to mean an instance of a concrete class which derives from *EDProduct*.

Event A concrete class. *Event* provides the interface used by *Module* code (among other clients) to obtain *EDProduct*s used for input, and also the interface to which *EDProduct*s are published.

Module Abstract base class of all the "worker units" manipulated *directly* by the framework.

EDProducer A *Module* which puts *EDProduct*s into the *Event*. Often, it will put only one; it is allowed to put more.

ModuleFactory A *ModuleFactory* creates *Module* instances.

Subsystem A *subsystem* is a loose collection of objects which act together to perform some clearly identifiable task.

Bibliography

[1] C. Grandi, D. Stickland, L. Taylor, *ed.*, **The CMS Computing Model**, CERN-LHCC-2004-035/G-083, CMS Note 2004-031.

[2] E. Frank, **ProxyDict Programmers Guide**, available at http://hep.uchicago.edu/~efrank/talks/ProxyDict.pdf