CMS Core Software Re-engineering Roadmap

the EDM design team

Revision 1.35

Contents

List of Figures

List of Unanswered Questions

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 that 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 usable (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 software 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, backgrounds, *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.

Insert a forward reference to 5.10 on page 22 here.

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 that 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 that match 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 Templating or Base Class for *EDProducts*?

It is possible to design the *Event* and related classes with either of two different different styles of design for *EDProduct*s:

1. *EDProduct*s all inherit from a base class, or
2. *EDProduct*s implement a simple generic programming *concept*.

In the first solution, users writing their data classes must inherit from a base class; in the second, there is no such need. In both cases, the “get” and “put” functions of the *Event* work with the user’s class directly.

In the first solution, heterogeneous containers of *EDProduct*s work because all inherit from the base class *EDProduct*. In the second solution, we’d have to introduce a base class, and a “wrapper” template that inherits from it, to form heterogeneous containers. This wrapper would appear as another level of indirection in the ROOT output files, and would be an (admittedly minor) annoyance to the user of the ROOT prompt.

Solution two would allow the use of simple things like `std::vector<CaloJet>` directly as an *EDProduct*. Solution one requires writing a class that could *contain* a `std::vector<CaloJet>`. This container would appear as another level of indirection in the ROOT output files, and would be an (admittedly minor) annoyance to the user of the ROOT prompt.

We want every *EDProduct* to know the *EDP_id* it is assigned. Solution one allows this functionality to be supplied by the base class. In Solution two, the “wrapped” *EDProduct* has this functionality, but the “bare” *EDProduct* does not.

The initial implementation of *EDProduct* used a base class. However, a decision subsequently was made to use wrappers, and an implementation using wrappers has supplanted the original implementation.

5.3 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.

5.4 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.

   The identifiers are unique to the event. It is possible to maintain common identifier lists and tag those with an ID and only record.

   Although a module can make use of more than one input to create its output, we make no attempt to specify the *type* of the *EDProduct* to which each of the entries in this vector refer. If such identification is needed in a particular *EDProduct*, that product can store the information in its own member data. We rejected providing a *mapping* of class name to *EDP_id* because we deemed the complexity unwarranted for the simple use to which the "parentage" information, in this general form, is put.

3. Executable configuration

   a) A “human friendly” string called a module label, which is a unique identifier (within a job) used for *EDProduct*s created by the module configured by this label. This label comes from a module configuration parameter with a fixed name. Each module has exactly one of these.

      The label configuration parameter is special. Changing the label in the configuration will cause a new module to come into existence because a unique ParameterSet determines module instances. However, the label is not part of the permanently generated ID.

   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.

   A pass identifier?

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).

5.5 Branch naming from provenance information

Provenance data will exist in a form that is traversable from the ROOT prompt (browsable).

The branch name is composed of a

1. Friendly product type name
2. Process name
3. Module label

The separator is assumed to be an underscore. Friendly product type name comes from the persistency tools. Its definition is assumed to be external to the source code and module configuration system. The provenance will not maintain this string.

This will help users move between the "data" and C++ object model views of the EDM.

5.6 Communication between event-processing elements

Approved on 7 March 2005

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, which is important for quality assurance of the physics results, we require that modules communicate *only* through the *Event*, by putting *EDProduct*s into the *Event*. Furthermore, 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 *EDProduct*s 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, for example to several output streams.

We recognize the fact that placing this requirement on the design of all *EDProduct*s makes the burden on the designer of each *EDProduct* class at least slightly greater. It is important for us to keep this burden as small as possible, without violating our other requirements.

5.7 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.

5.8 Event mixing module

5.8.1 Use cases

5.8.1.1 Use case 1

Merge Hits from a number of pileup simulated events to a signal simulated event.

The number of pileup events to be merged is a function of the machine luminosity. Out of time pileup must be considered in addition to in-time pileup, the number of crossings before and after the nominal one to be considered is sub detector dependent.

This is the main use case, as is needed for the digitisation of simulated data in the new framework. This usecase has a production flavour.

5.8.1.2 Use case 2

The framework should also be able to support, with a minor priority, the merging of a real data event to another real data event, to simulate more complicated events.

The merging of real data implies merging at the digi level, in contrast to the main use case. As the digitisation will have occurred in both data streams, the digi values cannot be added without an approximation or without full deconvolution of the digitisation followed by redigitisation.

5.8.2 Requirements

- there is an I/O efficiency requirement, at least for usecase 1. Considering out-of-time pileup from -5 bunch crossings and up to +3 bunch crossings and the highest luminosity giving 17.5 pileup events in average per crossing, this gives a number of 160 pileup events as input to be merged to a single signal event. Typical size of pileup events at Hit level is 150kB, and 500kB for signal events. That means that the mixing module must be able to handle 25MB of input and 0.5MB of output in a very short processing time.
- Fixed and Poissonian are different possibilities of calculating the number of pileup events to merge, all of them should be implemented.

- Merging of MC true information must be considered. A minimum here is to combine the MC truth at particle level, keeping track of which particles come from the primary stream (signal) and which come from the secondary stream (pileup or secondary signal).

5.8.3 Tasks of the mixing module

The mixing module closely collaborates with a specialised secondary input service in order to fulfill the following 2 tasks:

5.8.3.1 Event mixing

The mixing module superposes events, it does not perform the digitisation but delivers superposed events to the detector dependent digitisation modules.
It calculates the number of events requested from its own configuration (Poisson random or fixed, number of bunch crossings before and after) and asks the secondary input service to deliver them. The digitisation modules will have to verify that the range of delivered bunchcrossings satisfy their needs (they might throw an exception otherwise).
The way the events are superposed depends on the data tier that is treated:

- on the hit level the hits from the superposed events are simply added to the list of hits of the main event.
- at the particle level (MC true information) it is a simple cumulation as for the hits.
- At the digi level (analysis use case), the mixing means that the digis of the secondary stream are added to the list if they do not already exist, or produce a modified digi if a corresponding one already exists from the primary stream.

5.8.3.2 Access to the mixing information

The clients (digitisation modules in the main use case) will need to be able to identify for each hit if it comes from the pileup or from the signal event, and to which bunch crossing it belongs.

5.8.4 Configuration

Configurable parameter names are suggestions, they must be coherent with configuration general naming.

5.8.4.1 Configuration of the mixing module

```
string type = "poissonian", "fixed" (default is poissonian)
```

```
double average_number  (default is 17.5)
int32 min_bunch (default is -5)
int32 max_bunch (defauls is +3)
```

5.8.5 Configuration of the secondary input service

It is assumed that the secondary input service is able to provide randomly selected events from the secondary stream. It might be useful for debugging purposes to be able to ask for events selected sequentially. It might also be useful while asking for a random sequence of events to be able to forbid that an event be selected twice. Hence configuration should look like:

```
bool random (default is true)
bool no_reuse (default is true)
```

5.8.6 Mixing module in new framework

Since the mixing module is writing to the EventStore, it will be a EDProducer. As for the other modules, input and the output of the mixing module is the main event, and the result of the mixing module will be a EDProduct added into the main event.

5.8.6.1 EDProduct

The result of the mixing module is an object of type CrossingFrame. This object contains

```
  int RunID;
  int EventID;
  std::vector<SimHit> signal;
  std::vector<std::vector<SimHit> > bunchcrossings;
  int firstcrossing;  //=min_bunch
```

This is a first proposition, not yet finalised.

5.8.6.2 Architecture of the Mixing module

The mixing module is somewhat special since it is not driven by a single source of events, but by 2 input streams.There will always be a distinction between the main event stream, delivering one event after the other, and the secondary input stream delivering events randomly on request.
The mixing module creates and manages itself the secondary input stream, not known to the other modules.

5.8.6.3 Demands on the secondary input service

- A specific method of the input service will be needed, allowing to get a vector of events the length of which has been specified by the caller.
- The events delivered must be selected inside the finite statistics of the pileup sample. If not specified otherwise, the secondary input service delivers events randomly, it is its own task to guarantee maximum efficiency for this.

5.8.6.4 Example of a corresponding Parameter Set

```
process PROD  = {

        source = PoolInputService {
                string fileName = "main.root"
                int32 MaxEvents = 2
        }

        source minbias = PoolInputService {
                string fileName = "pileup.root"
        }

        module out = PoolOutputModule {
                string fileName = "CumHits.root"
        }

        module mix = MixingModule {
          source input=minbias
          string type = ``fixed''
          double average_number = 14.3
          int32 min_bunch = -5
          int32 max_bunch = 3
        }

        path p = { mix, out }
}
```

5.9 Schedule Specifications for the HLT

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

5.9.1 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 *EDProduct*s in the *Event*.

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

5.9.2 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.

5.9.2.1 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.

5.9.2.2 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.

5.9.3 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:

- $A \rightarrow B_1 \rightarrow C \rightarrow D \rightarrow F$ for t_μ

- $A \rightarrow B_2 \rightarrow C \rightarrow E \rightarrow G$ 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.

5.9.4 A Sufficient Solution

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

We see the need for a configuration language richer than the simple one above, that 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" that do not change the meaning of the program, but that avoid redundant execution of any module.

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

A trigger job will consist of many of these *TriggerTerms*.

5.10 Multiple Products from One *EDProducer*

Many problems are arising when we consider allowing multiple objects of the same type to be produced by a single *EDProducer*. The source of the problem is the fact that the produced objects are distinguishable neither by type nor by provenance (which describes the configuration of the producer and the "context" in which the producer was run). Thus, to distinguish between multiple objects of the same type produced by a single *EDProducer*, one is forced to look at the data of the *EDProduct* itself.

```
TriggerTerm    ::= PathExpression

PathExpression ::= PathExpression & Sequence
                 | Sequence

Sequence       ::= Sequence , ProcessingUnit
                 | ProcessingUnit

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

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

The problems include:

1. Schedule validation becomes difficult. It seems to require creation of prototype instances of the *EDProducts* at configuration time, so that the relevant data can be matched (by a *Selector* that knows about that specific *EDProduct*).

2. The need to create these prototype objects limits our flexibility in having *EDProducer*s announce what they make.

3. It requires that we support *Selector*s that look at *EDProducts* (and concrete subclasses), not just *Provenance*s. This puts a greater demand on the authors of *EDProducer*s and *EDProducts* to create the relevant *Selector*s. Previous experience leads us to believe it will be difficult to assure all *EDProduct* designers will produce the appropriate *Selector* classes.

We propose that *Selectors* passed to the *Event* should only operate on *Provenance*s.

When combined with the requirement that *EDProducer*s use only the `Event::get` function that returns a single *EDProduct*, there is a drawback to this choice. It means that *EDProducers* can not make use of the output of other *EDProducer*s that make multiple instances of the same type. We propose that the solution for this is for the *Provenance* to carry a user-supplied bit of data that can then be used by the *Selector* to identify a single matching *EDProduct*. However, we expect this to be a rare case.

5.11 Access to *EDProducts*

5.11.1 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 does so by examining the *Provenance* associated with that *EDProduct*. It encapsulates the user's requirements (*e.g.*, features of interest) for products. The typical use case will be to have a *Selector* examine a single feature and then through composition and more complex *Selector* can be created.

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.

5.11.2 This needs a home nearby

A note concerning selectors: In order to make selectors friendly to use, several standard selectors are necessary. A special GetLatestCreated (implies that the order of creation is maintained). The list used by this selector, is maintained as file-level metadata.

5.11.3 Guidelines

We believe the following guidelines are important:

1. It is convenient to allow human-readable strings (labels) to be used in the identification of *EDProduct*s.
2. It is critical that access using any such strings never anything other than a single *EDProduct*.
3. Access to *EDProduct*s must be type-safe, and so they must specify the (C++) *type of EDProduct* to be returned. Allowing the user to obtain all objects of a type that derive from a common specified type would complicate the interface for doing so. We believe such use would be rare.

See also § 5.5 on page 16 on branch naming, which touches on related issues regarding string labels.

5.11.4 Anticipated Access Needs

We the following access methods are important. The order below does *not* imply an order of importance. In each example below, the query supplies the type of *EDProduct* to be retrieved. This is only a representative sample, not an exhaustive list.

1. Return handles to all *EDProduct* objects of a given concrete type. This can return multiple handles. If no objects are found, throw an exception.
2. Return a handle to the one *EDProduct* object that has a given *EDP_id*. If there is none, throw an exception.
3. Return a handle to the *EDProduct* of a specific type, made by a particular reconstruction module, giving the full specification of the module's configuration.

In addition to these basic queries, we also need to support "modifiers" for queries—that is, the ability to select the *EDProduct*s to be returned by logical "and"-ing of the queries above with the following additions.

1. Restrict the *EDProduct*s returned by specifying a "process name".
2. Restrict the *EDProduct*s returned by specifying a "release number".
3. Restrict the *EDProduct*s returned by specifying some part of the *Provenance* information that must also be matched—for example, to obtain all the results of midpoint-cone jet algorithms, disregarding the remaining details of the configuration information.
4. Return the *EDProduct*s returned by specifying some feature (a value or range of values) of a parameter in the *ParameterSet* of the product's *Provenance*. For example, to obtain all the results of midpoint-cone jet algorithms that have a cone radius between 0.5 and 1.0.

Note that many *EDProduct*s contain *collections* of objects, *e.g.*, tracks, jets, or muons. The *Event* interface only supports obtaining *entire EDProduct*s. It does *not* directly support obtaining subsets of elements contained within and *EDProduct*.

5.11.5 Proposal

We propose that *EDProducts* should be accessed only through the *Event*.

We propose that all access methods will "match" only the *most derived type* of the *EDProduct* requested—that is, the type must be an exact match, not a match to a base class. One can not make any single "get" to obtain all *EDProduct* instances of types that derive from a common base.

We propose the *Event* has three member templates for retrieving *EDProduct*s. Each is parameterized on the type of the *EDProduct* to be retrieved.

1. `getBySelector` takes a *Selector* object and returns the single *EDProduct* that has *Provenance* that matches the *Selector*. If there is not exactly one such match this `get` throws an appropriate exception.
2. `getByLabel` takes a string label and returns the unique *EDProduct* of the appropriate type with a *Provenance* with the matching label. If there is not exactly one such match this `get` throws an appropriate exception.

3. `getByType` returns the unique *EDProduct* of the appropriate type. If there is not exactly one such match this `get` throws an appropriate exception.

4. `getMany` takes an optional *Selector* and returns all the *EDProduct* instances of the appropriate type that match the *Selector*. *EDProducer*s shall not use `getMany`.

5. `getManyByType` returns all the *EDProduct* instances of the appropriate type. *EDProducer*s shall not use `getManyByType`.

5.11.6 Consequences

We propose that the *ParameterSet* that configures an *EDProducer* must have a string parameter named "label". This will the the unique label assigned to every *EDProduct* created by the *EDProducer* instance configured with that *ParameterSet*.

At module configuration time, the uniqueness of these labels—for *EDProduct*s to be created by the current program—will be verified.

Because it is possible for the input file to contain *EDProduct* instances with labels that will collide with those of *EDProducts* to be generated in the current program, the system must have a way to deal with such collisions.

We propose that an attempt to insert a new *EDProduct* with label x into an *Event* that already contains an *EDProduct* of that type and label x result in an exception throw. This assures that the *Event* never gets into a state in which it contains two *EDProduct*s of the same type that have the same label.

In order to make it easy for users to run a reconstruction program that replaces "old" versions of some *EDProduct* with "new" versions of those *EDProduct*, we propose there exist an "*EDProduct* dropping module". This module should be user-configurable, so that it is easy for the user to specify which *EDProduct*s are *not* to be read from the input. This does *not* imply that all *EDProduct*s must be read from disk on input. Rather, it is possible that this "module" work by indicating to the file-reading mechanism that a given *EDProduct* "branch" is to treated is if it did not exist.

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 *EDProducer*s, 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 con-*

structed object (if any) and an error report object (if any). Only one of the two items would actually appear in any entry.

5.12 Insertion of *EDProducts*

See § 5.11.1 on page 23 on page 23 for the definitions of important terms in this section.

The question we answer here is: What code developers of *EDProducer*s write to put the *EDProduct*s they create into the *Event*?

We have two proposals, and need to select which one we will choose for further design.

5.12.1 Solution One

Data products are allocated and inserted through the *Event*.

The *Event* contains a `put` member template. `put` is parameterized on the type of *EDProduct* and is used to allocate an object that will automatically be placed permanently into the *Event* when the *EDProducer* successfully completes its task.

Each *EDProducer* that make more than one *EDProduct* will invoke `put` for each of the *EDProduct*s it generates. All the *EDProduct*s will be successfully added to the *Event*, or none will be added.

A *Provenance* object is added to the *Event* by the framework during the final commit. The user is not responsible for handling this information.

The main goal of this solution is to mimic the procedural model that we (Jim and Marc) have observed to be the favorite of most authors of reconstruction code.

5.12.2 Solution Two

*EDProduct*s are allocated dynamically and returned by the `produce` method of the *EDProducer*. Ownership of the created products is passed during the return.

The framework handles inserting the created products into the *Event*. A *Provenance* object is added automatically by the framework. The user is not responsible for handling this information.

The main goal of this solution is to allow the system to determine, at compile time, what *EDProduct* types are created by a given *EDProducer* by working with the return type of the `produce` function.

Thus the function type of `produce` is different for each kind of *EDProduct*.

5.12.3 Comparison

Solution 1 has "parallelism of interface". The user's model of the *Event* as repository of *EDProducts* is respected: a user obtains input *EDProducts* directly from the *Event* and inserts newly made *EDProducts* directly into the *Event*. Solution 2 lacks this parallelism.

In Solution 1, the transactional model is less obvious for *EDProducers* that make more than one *EDProduct* instance. The user's code might have the `put` for the two *EDProducts* widely separated. In Solution 2, the two objects are clearly returned together. Since the transactional model assures that *both* are committed or *neither* is committed, the behavior in both cases is the same.

The *Event* is already a heterogeneous collection of *EDProducts*. In Solution 1, we make use of this directly in the interface of the *Event* for both all `get` templates and for `put`. In Solution 2, we do not benefit from this, because the user does not put results directly into the *Event*. We see two different paths to follow:

1. Introduce a second heterogeneous collection of *EDProducts* to be the return type of the `produce` function, so that each function can have the same return type. This seems to violate the purpose of Solution 2, since it is no longer possible to deduce the type of the returned *EDProducts* from the type of the `produce` function.
2. Introduce several member templates (*e.g.*, `produce`, `produce2`, . . .), each of which has a different return type.

We expect that most *EDProducers* will insert a single *EDProduct* into the *Event*. Complicating the mechanism by which this is done for the common case, in order to support the rare case of multiple insertions, is a disadvantage.

5.13 Comparison of the *RecAlgorithm* System to *EDProducers*

In this section, we compare the initialization of *EDProducers* and *RecAlgorithms*.

5.13.1 What is an "algorithm"?

In the *RecAlgorithm* system, an algorithm objects are "entities that create new resulting objects or transform input objects". Algorithms in general

- have a state (or configuration),
- have some inputs, some of which are the results of other algorithms.

In the *EDProducer* system, we distinguish between

- *EDProducers*, the top-level, reconstruction-developer written, framework components that perform a quantifiable step of reconstruction, by creating and storing persistable data in the *Event*, and

- the much looser category of "algorithms", that can perform smaller steps of reconstruction, and that may be composed to form an *EDProducer*.

The *EDProducer* system has no formal notion of "algorithms" as components; the author of an *EDProducer* is free to use any internal organization suitable for his task.

5.13.2 Who are the "users"?

We find it useful to distinguish between two groups of "users":

- (algorithm) *developers*, who write reconstruction code (but are users of the core infrastructure), and
- *end users*, who configure and run an event-processing program.

Of course, an individual may play both roles, at different times.

5.13.3 Coupling of algorithm system to configuration system

In the *RecAlgorithm* system, the algorithmic objects are tightly coupled (though inheritance of implementation) with the parameter set (and other configuration) objects. It is not possible to make use of an algorithmic object without the presence of the full configuration system.

In the *EDProducer* system, an "algorithmic object" (*i.e.*, a component with an *EDProducer*) does not need to have any interaction at all with the *ParameterSet* object, nor with the rest of the parameter set system. The algorithm developer is free to decouple the algorithm objects from the parameter set system, if he so wishes. Because the *ParameterSet* system allows nesting of *ParameterSet* objects to arbitrary depth, the developer is free to make use of *ParameterSets* in configuring nested algorithms.

5.13.4 Algorithm names

In the *RecAlgorithm* system, each algorithmic object has a developer-assigned name; the examples show these names compiled into the code. The scope of these names is the entirety of ORCA; each algorithm name must be unique within this scope.

In the *EDProducer* system, the approximate equivalent to this algorithm name is the combination of *EDProducer class name* and *instance label*. The class name is of course unique within any program. The scope of an instance name is a single running process; the name must be unique within that scope. The instance label is a configurable parameter assigned (by the end user) at runtime. See elsewhere in this document for the use of the instance label.

5.13.5 Version identification

In the *RecAlgorithm* system, each algorithmic object has a version identifier, expressed as a string assigned by the algorithm developer in the code.

In the *EDProducer* system, the approximate equivalent is the CVS tag for the entire code-base used to construct a release; this is provided automatically, and is recorded in the created event data.

Developer-updated version information may be neglected by the developer, and is not certain to be of the "accepted format". Enforcement of standards is left up to inspection of releases. Automatic handling of the version updating can not be neglected, and is assured of uniformity.

5.13.6 Types of parameters

In the *RecAlgorithm* system, the *ParameterSet* supports types double, int, string, and bool. Additionally, the *RecConfig* that contains a *ParameterSet* can also contain additional *RecConfig*s.

In the *EDProducer* system, the *ParameterSet* supports these types, as well as vectors of these types. This *ParameterSet* also directly supports nesting of *ParameterSet*s, and vectors of *ParameterSet*s.

5.13.7 Handling of `double`s

In the *RecAlgorithm* system, each parameter of type double is associated with a *tolerance*. Such parameters are compared for equality using their tolerances.

In the *EDProducer* system, parameters of type double are compared directly. Note that such parameters are read, written, and compared—they are not used in calculation, and so no troubles with inexact representation of calculated quantities arise.

We note that use of tolerances in "equality" comparison cause the "equality" comparison to fail to satisfy transitivity. Failure of transitivity may lead to unexpected results.

5.13.8 Default parameters for algorithms

In the *RecAlgorithm* system, each algorithmic object is required to have a default value for each parameter compiled into the source code.

In the *EDProducer* system, to aid in tracing the behavior of reconstruction, default values of parameters compiled into the source code are expressly forbidden.

5.13.9 Calibration data in algorithm configuration

In the *RecAlgorithm* system, one of the parts of a *RecConfig* object is a *RecCalibrationSet*.

In the *EDProducer* system, calibration data are not considered to be part of the algorithm configuration system. They are handled as separate problems, and are decoupled.

6 Design of the Core Infrastructure

6.1 Overview

The main elements of the core infrastructure are:

1. the classes *Event* and *EventPrincipal*,
2. the "module" base classes *EDProducer*, *EDFilter*, *Mixer*, and *OutputModule*,
3. the classes *ScheduleBuilder*, *ScheduleExecutor*, and the variety of "worker" classes,
4. the *ParameterSet* class and its related classes,
5. the *Framework* class.

6.2 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 § 6.3.3 on page 37).
- 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 § 6.5 on page 38) 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*.

6.3 *EDProducts*

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

EDProduct is the base class for all objects inserted into the *Event*. Derived classes are also referred to as *EDProduct*s. Each instance of such a class represents a component of an event, and must be capable of persistence.

Each *EDProduct* instance has an ID that is unique within the *Event*, and which is assigned by the *Event* during its insertion into 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 *EDProduct*s 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.

6.3.1 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 § 6.5 on page 38). 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.

6.3.2 Rules for defining an *EDProduct* class and its components

Restrictions are imposed on each *EDProduct* and its components (i.e. members or base classes) of an *EDProduct* in order to assure:

1. *persistency*: Persistence capability
2. *browsability*: Browsability in a ROOT tree
3. *modularity*; Independence of the *EDProduct* or component from the algorithm that creates it, and from all other such algorithms
4. *reproducibility*; Reproducibility of results
5. *simplicity*; Avoiding unnecessary confusion
6. *implementation*: An implementation choice.

Each restriction is justified by one or more of these criteria, which are given before each restriction.

All classes used as an *EDProduct* or as a component (i.e. member or base class) of an *EDProduct* must meet the rules for an *EDM compliant class*. Transient components are not exempted. These rules are:

1. *reproducibility*; An *EDM compliant class* may not contain a `static` data member, unless the `static` data member meets all of these requirements:

   a) It is declared `const`.

   b) It is of integer type (e.g. `int`, `unsigned`, `long`), or a `struct` containing only integer types.

   c) It is initialized only with compile time integer constants (e.g. 4, -12).

2. *reproducibility*; A member function of an *EDM compliant class* may not contain a `static` data object, unless the `static` data object meets the same requirements that a `static` data member must meet (see previous item). n

3. *reproducibility*; An *EDM compliant class* may not contain a data member that is a `set`, `map`, `multiset`, or `multimap` where the key is or contains a floating point type (e.g. `float`, `double`) or a pointer.

4. *modularity*; Every *EDM compliant class* (or its template) must be defined in an "*EDM compliant header file*" (see below).

5. *modularity*; An *EDM compliant class* (or its associated header and source files) may not contain or depend on the algorithms used to create the class. A class used to implement such an algorithm, and the file defining it, is *a priori* not an *EDM compliant class*.

All classes used as an *EDProduct* or as a non-transient component of an *EDProduct* must meet the additional rules rules for an *persistence capable EDM compliant class*. These rules are:

1. *reproducibility*; A *persistence capable EDM compliant class* may not contain a non-transient `mutable` data member.

2. *reproducibility*; A *persistence capable EDM compliant class* may not contain a non-pointer transient data member, unless that data member can be deterministically initialized in every constructor of the class using nothing other than the values of persistent members of the class.

3. *reproducibility* A *persistence capable EDM compliant class* may not contain a transient data member that is a C++ pointer, or a container of C++ pointers, unless management or architect approval is received, AND all the following requirements are met:

a) No pointer may at any time point to anything external to the *EDProduct* in which it is contained.

b) Each pointer is deterministically initialized in every constructor of the class (possibly to 0 or an empty container).

4. *reproducibility browsability*; A *persistence capable EDM compliant class* may not contain a non-transient data member that is a C++ pointer, or a container of C++ pointers, unless management or architect approval is received, AND the following requirements are met:

   a) The type of each pointer must be that of a pointer to a class. Pointers to built-in types or pointers to pointers are not permitted.

   b) Each pointer must be the sole owner of the object to which it points. In other words, each pointed to object must be created either when the pointer is created or when the pointer is assigned, and each pointed to object must be destroyed either when the pointer is destroyed or when the pointer is zeroed or reassigned.

   c) If a non-transient T* is used, all the rules for components of an *EDProduct* apply to the class T and to the actual class of the pointed to object.

   There are two reasons that the user might wish to use a pointer to an object rather than to the object itself:

   a) If the object need not be present, a zero pointer can be used to indicate that no object is present.

   b) To implement polymorphism. However, polymorphism destroys browsability in that only the portion of the pointed to object corresponding to the declared pointer type may be browsed by ROOT. For this reason, we strongly discourage designs that use polymorphism in persistence capable objects.

5. *reproducibility browsability*; A *persistence capable EDM compliant class* may not contain an `std::auto_ptr`, `boost::shared_ptr`, `pool::Ref`, ROOT `TRef`, or any other "smart" pointer, with the following exception:

   a) `boost::value_ptr<T>` is permitted, where T is a class (not a built in type). The use of `boost::value_ptr<T>` instead of a T should be used only when the ability of `boost::value_ptr<T>` to specify the absence of a T is needed. When a `boost::value_ptr<T>` is used, all the rules for components of an *EDProduct* apply to class T. Note that `boost::value_ptr<T>` has value semantics, so copying it copies the T object.

   Note that the EDM will provide one or more *persistence capable EDM compliant class* that implements references or vectors of references.

6. *persistency*: Every *persistence capable EDM compliant class* must contain a default constructor.

7. *persistency*: Every *persistence capable EDM compliant class* must have a data dictionary.

The following additional rules apply to each class used as an *EDProduct*, but not necessarily to a class used only as a component of an *EDProduct*: (Note that the great majority of *EDProducts* are collections. For example a Jet is not an *EDProduct*, but a collection of Jets is an *EDProduct*.)

1. *implementation*: Every class used as an *EDProduct* must publicly inherit from the class `EDProduct`. The inheritance need not be direct. (see § 5.2 on page 13).
2. *modularity*; A class used as an *EDProduct* or any of its base classes must not directly inherit from more than one base class, not counting any base class that is a pure interface class (i.e. has no data members, either directly or by inheritance). Exceptions to this requirement may be granted by management or the architect.
3. *persistency*: The data dictionary entry for a class used as an *EDProduct* requires a unique ID. (This requirement may be removed when and if Pool fixes the the problem that causes the ID to be needed in some cases.)

The rules for an *EDM compliant file* (header or source) are as follows:

1. *simplicity*; An *EDM compliant header file* must have the same name (with .h appended) as (one of) the classes or templates defined in the header file, other than a nested class or template.
2. *simplicity modularity*; If a class defined in an *EDM compliant header file* has any member functions or associated non-member functions that are declared but not defined in the header file in which the class is defined, these functions, if defined, must be defined in a single associated *EDM compliant source file* (see below). The *EDM compliant source file* must have the same base name as the corresponding header file, with .cc appended, and be in the same package as the header file.
3. *modularity*; An *EDM compliant file* must not define a class or template that is not used as an EDProduct or a component of an EDProduct.
4. *modularity*; An *EDM compliant source file* must contain an `#include` of its associated header. An *EDM compliant header file* must contain a `#include` of any header needed to fully define a direct component of any class or template defined in the header file. Any such included header must also be an *EDM compliant header file*. Other than those headers, the only permitted `#include`s in an *EDM compliant file* are:
   a) C++ standard headers, as needed (e.g. `<vector>`)
   b) C standard headers, but only if no comparable C++ header exists
   c) Headers defined in the framework, boost, CLHEP, and other permitted external or internal low level packages, as needed. The complete list of such permit-

ted packages will be expanded as needed. POOL, SEAL, or ROOT headers are strictly forbidden.

d) Any headers that contain only forward declarations.

An `#include` of any other files are forbidden. Note that any headers defining algorithms for *EDProduct* creation may NOT be included in the headers or source files defining the *EDProduct* or any component.

The rules affecting packages (i.e. shared libraries) are as follows:

1. *modularity*; More than one *EDProduct* or component (i.e. *EDM compliant class*) may be defined in the same package. However, any package defining one or more *EDM compliant class* may not include any definitions of non-compliant classes (e.g algorithm classes).
2. *modularity*; The data dictionary for a CMS defined non-templated *persistence capable EDM compliant class* must be defined in the same package in which the class is defined. The data dictionary for a CMS defined templated *persistence capable EDM compliant class* must be defined in the same package in which the class is instantiated. This implies that every persistence capable templated class must be explicitly instantiated in one well-defined package.
3. *modularity*; If X is an *persistence capable EDM compliant class*, data dictionaries for classes such as `std::vector<X>` and `boost::value_ptr<X>`, if needed, must be defined in the same package as the data dictionary for class X. It is recommended that a dictionary for `std::vector<X>` be defined for every *persistence capable EDM compliant class*.

6.3.3 Rules for modification of an existing *EDProduct*

- 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*.

6.4 *Provenance*

A *Provenance* serves to collect the relevant information describing *how* a given *EDProduct* was created. Each *EDProduct* is associated (in an *Event*) with *one Provenance*.

6.5 *Selectors*

*Selector*s 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

6.5.1 Different selection of event products

The most general selection should provide as result more handles to selected *EDProducts*. 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.

6.6 Modules

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

6.6.1 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 *ScheduleExecutor*. 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.

6.6.2 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 6.6.5 on page 42.

6.6.3 *EDProducers*

Figure 2 shows a coarse view of the processing flow for the execution of a single *EDProducer*. The main purpose of this diagram is to demonstrate the call sequence, so function call arguments and object types are not present. In addition, the method names are only meant to denote the types of actions that occur during the calls. The user code never interacts directly with the actual event object (depicted in the diagram by “ep”). The *Event* object lives only as long as the call to the module and its primary purpose is to track objects retrieved from and created by the *Module*. Upon successful return from the user code, the *Event* generates all the proper provenance information for the newly formed *EDProduct*s and placed them all into the actual event object. If an exception is thrown anytime before this final commit to the event object, no new *EDProduct*s are recorded.

Figure 2: The flow of control for the execution of a single *EDProducer* object.

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 *EDProduct*s 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 responsibilities 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.

6.6.4 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* that is passed back to the framework. This is its only purpose.

6.6.5 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.”

6.7 The *ParameterSet* System

6.7.1 *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`
- `int`
- `std::vector<int>`
- `unsigned int`
- `std::vector<unsigned int>`
- `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 method of managing untracked parameters within parameter sets. These are similar to tracked parameters in how they are presented to the user, but they do *not* contribute to the parameter ID, and are *not* tracked in any repository. They are to be used to carry information that 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.

6.7.2 Identifying Parameter Sets

There will be a central authority to store those *ParameterSets* used in official processing. Each program will have access, in addition, to a local repository of *ParameterSets*, 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 (more precisely, from a string generated 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.

Unanswered Question 1: Is MD5 suitable for use as *ParameterSet_id*?

> The MD5 algorithm, proposed as the means of generating a *ParameterSet_id* from the contents of a *ParameterSet*, does not *guarantee* that differing parameter sets will have different *ParameterSet_id*s. There is a nonzero (albeit very small) probability that two different parameter sets will yield the same MD5-based *ParameterSet_id*. Is the degree of certainty of non-collision of two MD5 checksums sufficient for CMS?
>
> The MD5 algorithm is described in RFC 1321, available at `http://www.ietf.org/rfc/rfc1321.txt`. A paper announcing how this algorithm was "cracked" is available [3]. This "cracking" is not of importance to CMS; we are not looking for cryptographically secure identification. We are concerned only with *accidental* collisions.

6.7.3 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.

These tools will *not* be available in the first release of the parameter set system.

6.7.4 Informal *ParameterSet* language specification

The configuration language used in the creation of *ParameterSet* objects has elements that map into framework concepts such as *process* (a single job) and *module*. These domain-specific elements allow for better validation to be done while parsing and allow the user to better express what the intended behavior of a job is. Figure 3 on the next page shows an example of the expression of a job in the configuration language.

Within the program, each section labeled with keywords *module*, *source*, or *ParameterSet* is turned into a *ParameterSet* object. The *process* section is special because of the additions of the *sequence* and *path* types, which are not part of the *ParameterSet* interface. A *process* can contain any number of *path* and *sequence* statements.

The *block* keyword indicates a collection of parameters that are made available for inclusion in any other section that will be mapped into a *ParameterSet*.

The *using* keyword indicates that names and values from another *ParameterSet* or block will be introduced (pulled into) the current *ParameterSet*. This facility allow one

```
process myjob = {
  source = FileBasedInputService{
    untracked string fileName = ``myFile'';
    untracked bool buffered = true;
  }
  block Common = {
    untracked int debug_level = +1;
    untracked string unknown_exception_action = ``die'';
    untracked string user_exception_action = ``skip'';
    untracked string framework_exception_action = ``skip'';
    double radius = 0.4;
  }
  ParameterSet splitmerge05 = {
    double frac = 0.5;
  }
  module cone5 = MidpointJetProducer {
    using Common;
    ParameterSet  split_merge = splitmerge05;
    double radius = 0.5;
  }
  module cone7 = MidpointJetProducer {
    using Common;
    double radius = 0.7;
  }
  module jetanalyzer = JetAnalyzer {
    string wanted = ``cone7'';
  }
  module otherthing = SomeModule {...
  }
  module jetoutput = PoolOutputModule {
    string fileName = ``MyOutputFile'';
  }
  module all = PoolTriggerOutputModule {
    vstring terms = {``term1'', ``term2''};
  }
  module some = PoolTriggerOutputModule {
    vstring terms = { ``term1'' };
  }
  sequence cones = { cone5,cone7 };
  path term1 = { cones,jetanalyzer,jetoutput };
  path term2 = { otherthing };
  endpath = {all,some};
}
```

Figure 3: Elements of a job configuration

to inject values into a *ParameterSet* from another. If the same type and name are found in the outer scope and inner scope, the outer scope takes precedence and overrides the value from the inner scope.

A primary *source* must always be present, so it is implicitly assigned the name “main_input”.

6.7.5 *ParameterSet* object interface features

The interfaces outlined in Figure 4 on page 46 illustrate essential features; the exact names and signatures are left to the library designer.

```
class ParameterSet {
   public:
   // retrieve value of tracked type XX with name name
   // need one for each of the supported types and for vectors<YY>
   YY getXX(string name) const;

   // same as previous gets, but for untracked values
   // here default values are allowed
   YY getXX(string name, YY default_value) const;

   // inject a name/value pair into this parameter set,
   // need one insert for each type XX and vector<XX>
   void insert(string name, XX value, bool is_tracked);

   // an important feature of the next method is getting
   // a list of names given a specific type, either by the
   // keyword name or C++ type or name
   vector<string> getNames(string of_this_type) const;
};
```

Figure 4: Essential elements of a pset and process objects

The *Path* object returned from the *Process* shall contain information consistent with the grammar specified in section 5.9.4 on page 22.

6.7.6 Mapping from input grammar to *ParameterSet* objects

Elements of the configuration file language are translated into *ParameterSet* objects, according to the following rules.

Translation from sections of a configuration file named *module*, *source*, and *process* requires adding new fixed name and type parameters to *ParameterSets* representing these things. Below are three of these mapping expressed in the parameter set language.

```
// initial specification
module cone5 = MidpointJetProducer {
  using Common;
  double radius = 0.5;
}

// maps to
ParameterSet cone5 = { /* jbk - is this pset name correct? */
  string module_label = ``cone5'';            // check name
  string module_type = ``MidpointJetProducer''; // check name
  double radius = 0.5;
  untracked int debug_level = +1;
  untracked string unknown_exception_action = ``die'';
  untracked string user_exception_action = ``skip'';
  untracked string framework_exception_action = ``skip'';
}
```

Figure 5: Mapping from *module* section to *ParameterSet*

```
// initial specification
source = FileBasedInputService {
  untracked string fileName = ``myFile'';
  untracked bool buffered = true;
}

// maps to
ParameterSet main_input = { /* jbk - is this pset name correct? */
  string module_label = ``main_input'';
  string module_type = ``FileBasedInputService'';
  untracked string fileName = ``myFile'';
  untracked bool buffered = true;
}
```

Figure 6: Mapping from *source* section to *ParameterSet*

Notice that the *sequence* statement in the *process* mapping in figure 7 on page 48 was incorporated into the paths themselves.

6.7.7 Event mixing problem

There are details in this section that still need to be finalized. An example is that of an input source that can be asked to cough up more than one event in one invocation. We currently do not have such an interface. The information below indicates that a source used for mixing is really the same type of thing as a standard job input source. This is not

```
// initial specification
process myjob = {
  source = FileBasedInputService {...}
  block Common = { .. } // note this does not appear below
  module cone5 = MidpointJetProducer {...}
  module cone7 = MidpointJetProducer {...}
  module jetanalyzer = JetAnalyzer {...}
  module otherthing = SomeModule {...}
  module jetoutput = PoolOutputModule {...}
  module all = PoolTriggerOutputModule {...}
  module some = PoolTriggerOutputModule {...}
  ParameterSet splitmerge05 = {...}

  sequence cones = {cone5,cone7}
  path term1 = {cones,jetanalyzer,jetoutput}
  path term2 = {otherthing}
  endpath = {all,some}
}

// maps to
{
  ParameterSet main_input = {...}
  ParameterSet cone5 = {...}
  ParameterSet cone7 = {...}
  ParameterSet jetanalyzer = {...}
  ParameterSet otherthing = {...}
  ParameterSet jetoutput = {...}
  ParameterSet all = {...}
  ParameterSet some = {...}
  ParameterSet splitmerge05 = {...}
  vstring allmodules = {``cone5'', ``cone7'', ``jetanalyzer'',
                        ``otherthing'', ``jetoutput'',
                        ``all'', ``some''}
  vstring allpaths = {``term1:cones,jetanalyzer,jetoutput'',
                      ``term2:otherthing''}
  string endpath = {``all,some''}
}
```

Figure 7: Mapping from *process* section to *ParameterSet*

a bad thing, it just means that the input source interface probably needs an additional method for producing a vector of events.

The mixing problem here refers to the process of taking each event from the main input stream of events and perturbing or augmenting a set of objects within the event from data from another stream. In other words, a special stream of MC events are are blended into a main event data stream by specialized *EDProducer*s. Here are the list of assumption about this process:

- Each subdetector will have a specialize *EDProducer* that does the mixing for that subdetector.
- Each of these mixer *EDProducer*s uses the same blending event to modify the main event.
- The blending events for a given set of mixers all come from a single source.
- The input is expected to produce a vector

The parameter set language contains specialized keywords for configuring mixing jobs. The entire mixing process is encapsulated in a module that appears as a single *EDProducer* to the framework scheduler. Figure 8 on page 49 show a sample configuration of a job that does mixing.

```
process myjob = {
  source = FileBasedInput {...} // main stream of events

  // source of minbias events
  source minbias = PoissonFileInput {
    string fileName = ``precious_data'';
    double poisson_mean = 14.3;
  }

  // the special modules that do the mixing
  mixer calmixer = CalorMixingModule {...}
  mixer pixelmixer = PixelMixingModule {...}

  // the module that holds everything together
  module mixall = MixerBlock {
    source input = minbias;
    vmixer mixers = { calmixer, pixelmixer }
  }

  module output = PoolOutputModule {...}

  path p = {mixall,output}
}
```

Figure 8: Sample mixer configuration

A *Mixer* is a module that receives the current event as an argument, along with a list of other events that need to be blended into objects held within the current event.

6.8 The *ModuleFactory* and *ModuleRegistry*

6.8.1 *ModuleFactory*

A phase-1 implementation of the ModuleFactory class has already been written, and is available in the CVS repository. We have not yet written it up here.

The ModuleFactory is used only by the ModuleRegistry; all other client classes that wish to obtain an instance of a configured module should use the ModuleRegistry.

6.8.2 *ModuleRegistry*

The *ModuleRegistry* provides a caching layer on top of the *ModuleFactory*. It is used by the *ScheduleBuilder*, among other clients, as the source for configured module instances.

The main member function of *ModuleRegistry* is `ModuleRegistry::getWorker`, which has the same argument list as `ModuleFactory::makeWorker`.

In addition, *ModuleRegistry* should provide the ability to iterate through all the currently-cached module instances.

6.8.2.1 Future Additions

In a later phase of development, but not in phase 1, the *ModuleRegistry* should provide version management and validation for 'plug-ins". Using "cvs tags", it should verify that all modules come from the same CMS software release. It must support a development mode, where untagged modules can be mixed in with released one, and assure that the provenance information clearly indicates this. This implies that the cvs tag string will probably need to be compiled into the plugin library by the build system. The SEAL *PluginManager* may already do this.

6.9 The Scheduler System

The scheduler system is the subsystem in the framework that is responsible for executing the sequence of reconstruction and decision making 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.

Figure 9 on the facing page shows the organization of *EDProducer*s and *FilterModule*s and their relationship to elements present in the configuration of the trigger as presented in § 5.9.4 on page 22. *FilterModule*s and *EDProducer* modules are treated similarly by the scheduling components as seen by the *Worker* abstraction. The main difference between the types of *Worker*s is that *ProducerWorker*s always "pass" the event and *FilterWorker*s reflect the value returned from the *FilterModule*.

Figure 9: Class diagram of the *ScheduleExecutor* and its relationship with various types of modules.

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 *ScheduleExecutor*.

Both *ScheduleBuilder* and *ScheduleExecutor* 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 *ScheduleExecutor* must assume that each *EDProducer* may request stopping of execution of that "path."

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

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

6.9.1 *ScheduleBuilder*

The *ScheduleBuilder* uses the parsed path expressions from a *ParameterSet* object to create a "schedule". The schedule is expressed as a sequence of "Workers".

The process of creating the sequence of "Workers" should have the following steps:

1. substitute sequence nodes into path nodes (sequences are just aliases);
2. verify that prerequisites as declared in path expressions are consistent as specified in the parsed file
3. remove redundency in each of the paths, and
4. build the sequence of "Workers" to be given to the *ScheduleExecutor*.

6.9.1.1 Prerequisite Consistency Check Algorithm

Prerequisite checking only verifies that names declared in paths to be dependencies for other names are consistent across all paths.

Here is one example of an algorithm that can check consistency of prerequisites expressed in paths and sequences. This algorithm requires using the output from the parser, which is a list of binary trees with operator nodes and operand nodes (leaves).

This algorithm also requires walking up the tree from a leaf node. The node classes that represent operators/operands may need to be modified to allow this.

```
1) substitute sequences into paths

   build map of sequence_name -> sequence node (WrapperNode)

   for each i in path/end_path
    for each element in i
     if i is a sequence, then
       substitute the sequence operator node into the tree
       and make parent adjustments

2) gather up all leaf nodes

3) Validate consistency

   make map of module_name -> dependency list object
   for each n in leaf nodes
    make new deplist object
    p = parent of n
    while p
     if p is an '&' operator, then continue
     if left operand of p is not the path we arrived on, then
      findDeps(left operand of p, new deplist to fill)
     p = parent of p

    sort deplist and remove duplicates

    attempt to insert the deplist into the map using name in leaf
    if failed
     if the entry in the map is not equal to the new one, then
      we have a prerequisite inconsistency for this leaf

4) findDeps( node, deplist )
     if node is a leaf, then same name in deplist
     else
       findDeps(left node of node)
       findDeps(right node of node)
```

6.9.1.2 Path reduction

After initial validation, redundancy within each path can be eliminated. This elimination can also be done strictly using the leaf names. For each path, form a list of leaf names using a depth first left to right traversal of the node tree. For each name in the list, remove all instances of that name that appear after this entry.

6.9.1.3 Future Additions

In a later phase of development, but not in phase 1, the following additional steps should be taken:

1. verify that prerequisite *EDProduct*s are available before each “Worker” is invoked, and
2. allow requests for reconfiguration of modules in a schedule.

6.9.2 *ScheduleExecutor*

6.10 The *EventProcessor*

6.11 Non-Event Data

The Non-Event data is data whose ’interval of validity’ (*IOV*) is longer than one Event. We have two types of *IOV*s which are distinguished by whether or not the DAQ system initiated the interval of validity transition. *IOV*s initiated by DAQ (such as the Event or a Run transition) are to be handled by the Event system. All other *IOV*s are handled by the *EventSetup* System.

6.11.1 *EventSetup* System

Figure 10: The *EventSetup* is formed from the Records that have an *IOV* that overlaps with the moment in time that is being studied.

The *EventSetup* provides a uniform access mechanism to all data/services constrained by an *IOV*. The main concepts for the *EventSetup* are:

1. Record: holds data and services which have identical *IOV*s.
2. *EventSetup*: holds all Records that have an *IOV* which overlap with the ’time’ of the Event being studied.

6.11.2 *EventSetup*

The *EventSetup* class provides type-safe access to the various *Records* it contains. This access is done through the *EventSetup*’s `get<RecordT>()` method. If the requested Record is not available, an exception will be thrown. There is also an interface to get data directly from an *EventSetup* instead of from a *Record* for the case where the data type has been assigned a ’default’ *Record*. The direct data access interface is discussed in a section 6.11.4.

In addition to access to *Records*, the *EventSetup* has a method `timestamp()` which return information about the 'instance in time' for which the *EventSetup* is describing.

6.11.3 *Records*

Records provide type-safe access to the objects it contains. The access is handled through the *Record*'s `get(ESHandle<T>& )` method. This is analogous to data access from the *Event*.

A *Record* also provides access to its interval of validity (*IOV*) through its `validityInterval()` method.

6.11.4 Contents of a *Record*

The *EventSetup* system sets no requirements on the C++ class type of an object which may be placed in a *Record*. The only restriction is the lifetime of the objects within a *Record* is only guaranteed to be as long as the *IOV* for which the *Record* is appropriate. This does not mean that an object within a *Record* can not be reused across an *IOV* transition, it only means code that reads the object from a *Record* should not assume that it will be reused.

In the case where a data/service C++ type is only meant to come from one Record type, then the 'default' Record type can be declared at compile time. If a 'default' Record has been declared for a data type, then users can access that data directly from the *EventSetup* via the `get(ESHandle<T>& )`.

6.11.5 *EventSetup* System Components

The *EventSetup* system design, shown in figure 11, uses two categories of components to do its work: *ESSource* and *ESProducer*. These categories are 'high-level abstractions' built on top of low-level interfaces.

Figure 11: The *EventSetup* system design.

6.11.5.1 Low-level Interfaces

DataProxies: When a `get` call is made to a *Record*, a DataProxy is looked-up in the *Record* and then that DataProxy is asked to handle the request.

DataProxyProvider: A container of related DataProxies. Advertises what *Records* it has DataProxies for and is capable of creating those DataProxies on request. The DataProxyProviders are given to the EventSetupProvider who hands them off to the proper EventSetupRecordProvider. The EventSetupRecordProvider then insert the DataProxies from the DataProxyProvider into the proper *Record* instance.

ProxyFactoryProducer: A DataProxyProvider which uses pre-registered Proxy factories to create the necessary DataProxies. This is a convenience class for developers who need to be able to write their own DataProxies.

ESProducer: The easiest to implement DataProxyProvider. Simply by writing a `produce` method, the *ESProducer* will use the argument to the method to determine the *Record* the algorithm is dependent upon and will use the return value to determine the type of data/service being created. This deduction is done by calling the `setWhatProduced(this)` from the class' constructor.

EventSetupRecordIntervalFinder: Interface for determining the proper *IOV* of a *Record*, or a group of *Records*. The EventSetupRecordIntervalFinders are given to the EventSetupProvider who hands them off to the proper EventSetupRecordProvider. The EventSetupRecordProvider uses its EventSetupRecordIntervalFinder to set the proper *IOV* of its *Record*.

6.11.5.2 EventSetup Source

An *ESSource* is responsible for determining the *IOV* of a *Record* (or a set of set of *Records*). The *ESSource* may also deliver data/services. An *ESSource* must inherit from EventSetupRecordIntervalFinder and if it is delivering data/services it must also inherit (possibly indirectly) from DataProxyProvider.

An *ESSource* normally reads its information from a 'persistent store' (e.g., a database) although it is not required to do so.

6.11.5.3 EventSetup Producer

Conceptually, an EventSetup Producer is an algorithm whose inputs to its algorithm are dependent on data with *IOV*s. From an implementation stand-point, an EventSetup Producer must inherit from DataProxyProvider.

6.11.6 Dependent Records

Sometimes an algorithm in the *EventSetup* is dependent on data coming from more than one *Record*. For example, the tracking geometry is dependent on the 'ideal geometry' and on the tracking alignment values. In such a case the *Record* used by that algorithm needs to be declared 'dependent' on the other *Records*. This dependency declaration is accomplished by having the dependent *Record* inherit from `DependentRecordImplementation<T,Lis` The template parameter `List` is a compile time list of the *Records* upon which this *Record* is dependent.

Dependent *Records* allow access to the *Records* to which they are dependent via the `getRecord<T>()` method.

The *IOV* of a dependent *Record* is the intersection of the *IOV* of all the *Records* to which it depends. The *EventSetup* system guarantees that the proper relationships between the *IOV*s is preserved.

6.11.7 *EventSetup* configuration

EventSetup components are configured using the same configuration mechanism as their *Event* counterparts, i.e., via the *ParameterSet* system. Two additional keywords were added to the configuration language to allow the configuration of the *EventSetup*: `es_source` and `es_module`.

6.12 Data Management

Commentary from: Luca Lista

The need for input and output modules is specified in section 6.6.5 on page 42. The main applications will use POOL data format to write and retrieve data. It would be convenient to allow multiple input and output modules to run concurrently in the same job; multiple input modules, together with an appropriate event mixing module, can provide the ability to mix simulated event with real minimum-bias background; multiple output modules allow the writing of 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* polymorphically, 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* that is used internally by the class *Event*, that provided polymorphic 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 convenient and efficient to use `markMultiwrite` on references to *EDProduct* selected to be stored; in presence of objects cross-references, in the cases where those

[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.

could be required, `markMultiwrite` does not guarantee 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?

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 that 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* that 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 that 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`

[3] X. Wang, D. Feng, X. Lai and H. Yu, **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD**,*Cryptology ePrint Archive, Report 2004/199*, available at `http://eprint.iacr.org`.