

# Preparatory Notes for the CMS Infrastructure Review

Ken Bloom  
Walter E. Brown  
Greg Graham  
Liz Sexton-Kennedy  
Jim Kowalkowski  
Marc Paterno  
William Tanenbaum  
Avi Yagil

November 29, 2004

## Abstract

This document discusses what the LPC Edm group considers best practice in the domains of Framework and Edm. It is informed by our Run II experience, and documents our current understanding of the CMS software in these domains.

## Contents

<b>1</b>	<b>Purpose of this Document</b>	<b>2</b>
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>The Task View</b>	<b>3</b>
3.1	Analysis task . . . . .	4
3.2	Reconstruction task . . . . .	6
3.3	Trigger task . . . . .	8
3.4	Instrumenting the Framework . . . . .	8
3.5	Running the Program . . . . .	8

3.6	Event Processing Schedule . . . . .	9
3.7	Dynamic Libraries . . . . .	10
3.8	Topics We Haven't Time to Write About . . . . .	11
<b>4</b>	<b>A Comparison Between CDF and CMS</b>	<b>14</b>
4.1	The Architecture View . . . . .	14
4.1.1	The Module Layer . . . . .	14
4.1.2	The Algorithm Layer . . . . .	17
4.1.3	The Objects Layer . . . . .	17
4.1.4	The Geometry Layer . . . . .	22
4.1.5	The Infrastructure and Services Layer . . . . .	22
4.1.6	Utilities and 3rd Party Packages . . . . .	22
4.1.7	PackageList . . . . .	22
4.2	EDM . . . . .	23
4.2.1	The Event . . . . .	23
4.2.2	Collections, Objects and the Associations Between Them . . . . .	25
<b>5</b>	<b>Metadata Management</b>	<b>26</b>
5.1	CDF Use of SAM . . . . .	26
5.2	Object-level Metadata and CMS . . . . .	27
5.3	Dataset-level Metadata and CMS . . . . .	27
5.4	Event-level Metadata and CMS . . . . .	28
<b>6</b>	<b>Summary</b>	<b>28</b>

---

# 1 Purpose of this Document

The purpose of this document is:

- to list many good concepts that should be present in a framework and event model based on Run II experiences;
- to provide (to the best of our knowledge) a list of features missing from the CMS environment;
- to establish a set of rules or guidelines for ORCA code developers, in order to help determine the interaction between the various physics packages and the framework and the structure of objects in the event data,

- to aid in the production of an mutually-agreed-upon list of changes that are needed—this could include changes in concepts and design of ORCA and COBRA;
- to aid in our production of a workplan to execute the above.

We would like to stress that our purpose is *not* to have CMS emulate CDF's design. The intent is to *identify the good features we rely on, independent of implementation details*. We also want to identify and learn from our failures or poor choices. We tried to represent both in our examples.

## 2 Introduction

Any large software system (greater than 20 packages contributing to the same binary) with a large number of people (greater than 10) has a tendency to become chaotic. It is hard for any one person to know where it is best to place his class with respect to the physical design of the large system. At CDF we give our developers some simple guidelines in order to make the system self-organizing. The goal is to maintain a list in which each package (which usually consists of one library) is only listed once, *i.e.*, there are no circular dependencies and the ordering of the list is completely *independent* of the application that is being linked. Developers are asked to identify to which layer of the architecture their class belongs. A layer consists of many different detector or physics packages that play a similar role in the architecture of the system. For instance in the module layer we have `ElectronMods`, `TopMods`, `CalorMods`, `TrackingMods` *etc.* After they identify the layer for their class, they place it in the correct sub-system package for that layer.

In the next section, various user tasks are discussed, with some comments on how CDF and DØ meet the demands of these tasks. In the subsequent sections, each layer of the architecture is described. The description is ordered from the top-most layer of the hierarchy (nothing can depend on top-most layer packages except the application builder) to the bottom-most layer (anything can depend on this package).

## 3 The Task View

In this section we present several tasks that a (physicist) user might want to do. The examples we have chosen are:

- analysis
- reconstruction
- trigger
- code behavior analysis

We do not intend to present formal *use cases*; we sketch only enough of a description of the task to make the necessary points clear. For each task, we describe the main features of the software of one or more experiments that we believe produced a successful solution. In some sections, we provide a short series of related tasks to illustrate different aspects of the experiments' solutions.

### 3.1 Analysis task

First consider a simple task of histogramming the transverse momentum of the leading  $p_T$  muon in each event. We start from a DST, and we do not want to perform new reconstruction. We want to find those muons, already reconstructed, that were identified by a specific version of the muon reconstruction algorithm and with a specific set of parameters used in the reconstruction and particle identification.

Each of the experiments we have worked with has the concept of a *framework module* whose purpose is analysis (as opposed to reconstruction, online filtering, or other tasks).

**Framework module:** A *framework module* a coherent body of code (an object) that operates on a physics event and responds to external stimuli (computer science “events”) to perform various actions related to its single task. It performs its task without direct interaction with other framework modules. It can make use of *framework services*, defined later.

*Framework  
module*

A user writing such a module would be responsible for implementing the few necessary member functions for the module to perform its task.

One such member function is the module constructor. After construction the module should be in a functional state. It should have correct values for all its parameters and should not require later re-configuration before use. The parameters for the module should *not* be compiled into the code; it should be possible to inspect the parameter set *without* having an instance of the module—or the source code for the module—present. There should not be a two-phase (or multi-phase) configuration in which configuration is *begun* on construction but is *completed* only after another module has been constructed.

The CDF framework has the concept of a help system that tells the users the meaning and allowed values of various parameters used to configure a module. Users have found this very valuable. The CDF mechanism is more tightly bound to the module than we prefer; we prefer that the help system knows about *parameter sets*, but not about modules. We prefer that the code of the module is independent of the code of the help system, and *vice versa*. DØ has the concept of configuring a module with a single set of parameter values; a module can be re-configured by giving it a new set of parameters. This organization makes the reconfiguration logic simple to write and easy to understand. The framework is responsible for associating a specific parameter set with the appropriate module instance.

In our example, the creation of the histogram in which we will collect muon  $p_T$  values should be done in the constructor.<sup>1</sup> The parameters for the histogram (the number of bins, and the minimum and maximum values for the range) should not be specified in the code, but should be provided by the parameter set given to the constructor. Within the constructor, we query the parameter set for these values. The parameter set has a type-safe interface for retrieving the value associated with a given named parameter. The parameter set conveniently groups associated parameters at a level of granularity chosen by the developer of the module.

The second important member function of an analysis module is the *analyze event* function. The argument for this function is (a `const` reference or pointer to) an *event*.

**Event:** An event is an in-memory object database that can support insertions and queries, but that supports neither deletions nor modifications of objects already inserted. The “query language” is not general, but is instead tailored to the physicists’ needs, and is expressed in the interface of the event class. Each event contains raw data and derived products (such as trigger output and reconstruction artifacts) related to a single beam crossing, or simulation thereof.

*Event*

In our example, we are considering the histogramming of muons. Let’s consider several sub-examples.

In one case, we want to histogram only those muons from a specific algorithm, in a specific code version, configured with a specific set of parameters. We want to make sure our sample is not contaminated by the event data objects that are the product of any other “rogue algorithms.”

**Event data object:** An *event data object* is either a part of the raw data, of the simulation information, or of the output of a reconstruction algorithm. It has a

*Event  
data  
object*

---

<sup>1</sup>The business of how to interact with ROOT is complicated, and we will avoid the details here.

unique identifier *within the event*.<sup>2</sup> Associated with each event datum (but not necessarily stored in the event) is an object that holds the provenance of that datum. Such provenance information includes the details of the configuration of the algorithm that made the event data object and the identifiers of other event data objects used as inputs for the reconstruction of the event data object. The provenance may include additional information. There is no “algorithm part” to an event data object.

In another case, we want to histogram the output of several algorithms, each into its own histogram. In this case, we don’t want to fix the number of histograms during construction of the module; we want to discover what algorithms have been run, and histogram their output, and record the configuration information for each algorithm from which we discover a reconstruction product.

In yet another case, we want to histogram the output of a specific algorithm, and want the newest approved-for-conference-use version of that algorithm found in the event, but reject versions that are “too old” or “too new.”

To support all these uses, the event data must be associated with the full set of configuration information, and the event’s query mechanism must be able to use all of, or any part of, that information as a constraint upon selection.

A third example member function is the *divulge statistics* function. This function serves as a signal to the module that it is time for the module to report its current state. Clearly a more detailed specification than this is necessary—we include it here primarily to illustrate that not all module member functions have to do with (physics) event processing.

Two additional module member functions are *end of run* and *end of job*. In our example we have no need of these functions, so we do not implement them.

## 3.2 Reconstruction task

There is a wide variety of types of reconstruction tasks. We will use as an example *missing  $E_T$  reconstruction*, because it seems to be among the simplest of reconstruction tasks.

We create a single module to perform this task. The constructor for this module is passed the same sort of parameter set object as was the analysis module from section 3.1. For each event, the algorithm must use a particular set of event object instances as input. In this constructor we specify *not* which actual instance of event objects are to be used as input—since they do not yet exist, and change from event to event—but rather how the algorithm is to identify which calculation of

---

<sup>2</sup>We see no need for this identifier to be *globally* unique and the cost of making it globally unique seems prohibitive.

calibrated calorimeter energies it will use and which vertex it will use. These specifications need to be sufficiently accurate to be unambiguously identify the event objects to be used as inputs for the missing  $E_T$  calculation algorithm. In each case we specify the type of the event data object and a description of the configuration of the module that created that object. During construction of the module, we also get a handle to any *framework service* we need—in this case the *calorimeter geometry service*, which we will make use of during reconstruction.

**Framework service:** To get access to a global resource, we use a framework service. Services manage initialization of, access to, and lifetimes of objects that provide non-event data—such as geometry information, and run conditions information. It may be that access to ROOT histograms, *etc.* should also be managed by a service. Our list is not exhaustive.

*Framework  
service*

For the analysis module of section 3.1 we discussed the *analyze event* function; for a reconstruction module we implement a *process event* function. The difference is that *process event* is passed a non-const reference or pointer to the event and is expected to modify the event by the insertion of a new event data object. In this function, we first get handles to the correct input objects: the container of calibrated calorimeter tower energies and the requested vertex. If either is missing, we create a missing  $E_T$  object that contains status information indicating that we have tried and failed to reconstruct the missing  $E_T$ , and also contains the reason for the failure.

If the required inputs are found, we then iterate through the collection of calorimeter tower energies (in the event data object we obtained above), and determine the directed energy vector from the vertex to the center of each hit tower, summing the  $x$  and  $y$  components. This requires use of the geometry service to determine the center of each tower in the collection of hit towers.

**Geometry service:** The *geometry service* is responsible for determining the identifier of the current run, for determining what survey information to use for the run, and for translating physical component identifiers to obtain geometry information about the identified detector component. It is independent of event data objects—it is possible to determine the number of  $\phi$  segments in the hadronic calorimeter without having any event data present.

*Geometry  
service*

At the end of the iteration, we create a missing  $E_T$  event data object, with the appropriate data values, and insert it into the event. This object is labeled with several pieces of metadata:

- the identifiers of the calorimeter tower energy and vertex event objects;

- the identifiers of the parameter set used to configure this module;
- possibly other items.

The missing  $E_T$  object is issued its own unique (within the event) object identifier upon insertion into the event.

In the design of the MiniBooNE reconstruction model, we found that it was possible to automate nearly all of this identification and labeling process. Authors of reconstruction algorithms need to do almost nothing in order to have their reconstruction products fully identified.

### 3.3 Trigger task

The trigger system places what are probably the strongest constraints on the scheduling features of framework. CDF has the concept of a *trigger path*, which consists of a series of modules in a fixed order that act together to produce the portions of event reconstruction necessary for a specific trigger decision. However, at CDF testing a trigger path in simulator is not sufficient to understand its behavior, because previous flows affect the result—because inputs are not sufficiently specified. Modules in different paths appearing in different orders could give different results. In §3.6 we address the subject of scheduling in more detail.

*Trigger  
path*

### 3.4 Instrumenting the Framework

It has often been useful to measure the performance of reconstruction programs—for example, to determine the speed of each individual task, or to find the location of a memory leak. The module-based framework has made this easy to do, because such tasks are handled in a common place and require neither instrumenting of user code nor recompilation of the program.

### 3.5 Running the Program

Users at both CDF and  $D\bar{O}$  have complained about the difficulty of running their respective reconstruction programs. Few users have expert knowledge of the program, causing many to use ntuple-form output. Both experiments have a system that is very flexible and powerful. However, both systems would benefit from more attention to ease-of-use: presenting the configuration in a simpler fashion, and guiding the user through more limited choices to make *reasonable* configurations of the program. In both experiments the configuration is hierarchically defined, and each node in the hierarchy can be defined in its own file. While this is a

good thing, it makes it difficult for the user to understand the result of the configuration. It is hard to know what combinations of modules are valid; the system does not help the user know what is valid. A browser of the hierarchy is a valuable tool; DØ has recently developed such a thing. But even this does not allow the user to understand what change in an “upstream” module will cause a change in the behavior of a “downstream” module. Both experiments record the configuration of the program at it was run, but both experiments lack a way to easily browse this configuration information, or to easily share it between users.

Both experiments suffer from a lack of distinction between the *build* environment and the *execution* environment. At DØ the execution environment was introduced as a late concept; tools (scripts) are provided to configure the environment and run the program. These tools are complex. At CDF there is no separation of the environments because a function (the constructor for the class `AppUserBuild`) is intended to be tailored by the user, in order to determine what modules are available for use, and what their instances’ names should be.

### 3.6 Event Processing Schedule

One of the important functions of a framework is to build an event processing schedule. The schedule expresses what activities can be done in parallel and what must be done serially. The “milestones” are also identified (multiple activities must be completed before continuing). Many of the concepts listed below are necessary for building a good schedule, one that:

- uses resources efficiently,
- minimizes event processing time,
- eliminates redundant calculations,
- is easy to configure,
- gives consistent and proper results, and
- can alert the user if a configuration is invalid or has ambiguities.

The sequences, dataflows and decision points, multithreading, and input and output requirements can all be used in the creation of an event processing schedule. Removing one of these pieces of information from the problem list will likely mean that the schedule will be suboptimal. Are the four items listed above enough to create this schedule at run time? How are they expressed in the configuration of the program?

Inconsistent results is one problem that can occur if there is not enough information available during the schedule generation. A simple example illustrates a problem that can occur.

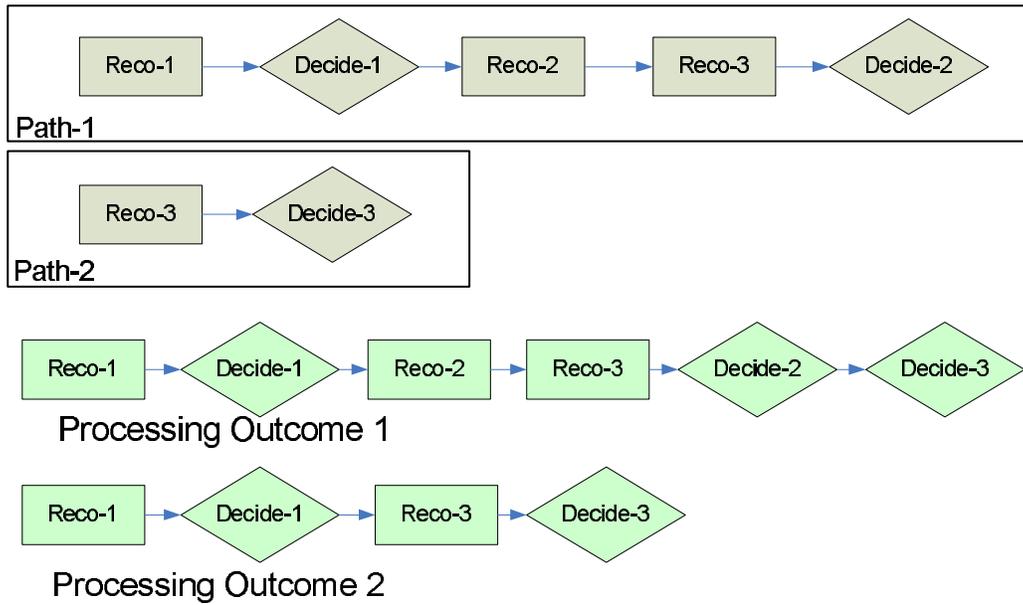


Figure 1: Example paths and possible execution sequences.

Figure 1 shows two paths that have the require the same instance of reconstruction to run in order to make a decision. The framework will ensure that this reconstruction only occurs once for each event. If the outcome of *Reco-3* depends on *Reco-2* results and this relationship is not enforced within the framework, then the outcome of *Decide-3* will have a hidden dependency on the outcome of *Decide-1*. To determine whether this set of paths produces a consistent result would require some kind of “coverage analysis,” where data representing the entire range of inputs are fed through and the results are checked for consistency.

### 3.7 Dynamic Libraries

We want to discuss the use of dynamic (shared) libraries.

Does dead (unused or test-related) code in a dynamically linked library cause performance problems because it is always present, unlike a static library where dead code costs little to nothing? It takes up virtual memory when the file is mapped. Is it true that a small amount of dead code is not really going to hurt

much because of the demand paging mechanism in the operating system and the lazy symbol linking?

Should shared libraries used by an executable come from only one frozen release so they are always consistent? Is this a written procedure or a policy actually enforced in the program? How does this policy or procedure fit in with algorithm developers? Is it too restrictive? Are there different rules for developers versus production? Can the build produce a dependency database that is built into the programs and also punch versioning tags (library and release) into libraries to be used for validation? A scheme such as this could allow the dynamic library mix to be determined by program configuration at run time.

How does one insure that in critical applications the program will not fail after a period of running because a shared object library cannot be loaded after encountering an event that causes a new event processing path to be executed? One method is to only pull in libraries during program configuration.

Where is explicit loading and implicit loading used? Are only low-level libraries such as ROOT and persistency explicitly linked? Are all the reconstruction tasks implicitly linked?

### 3.8 Topics We Haven't Time to Write About

There are many topics we believe we need to discuss during our meeting, but about which we do not have time to write. We list them here with brief explanations, to help establish the scope of topics that we believe it is important to cover. We have ordered them in order of their importance, with the most important items first.

1. **INTER-OBJECT LINKS:** We believe it will be advantageous to make sure that the persistent form of inter-object pointers within an event is realized in terms of "dumb data" rather than any persistence mechanism's smart pointers. The CDF and D0 experiments placed restrictions on the use of pointers within persistent objects. Objects in an event could refer to other objects in the same event or to items heard within them. A reference to an item within an object appears as a tuple of identifiers (EDOID,index), where EDOID is the event data object ID discussed earlier in this document and index is the item in this object in which we are interested. The event data object that supports indexing is required to supply a method that will produce the desired object given the index. Many persistent data objects fit this random access container pattern.
2. **SEQUENCES OF MODULES:** We believe it is useful for the framework to have a concept of *sequences* of modules that can be manipulated as a single unit.

This organizational component helps reduce the complexities of understanding what a particular program configuration is doing.

3. **DATAFLOWS AND DECISION POINTS:** We believe that, in conjunction with sequences from item 2 above, that it is useful for the framework to have the concepts of *dataflows* and *decision points*, from which can be constructed trigger paths (among other purposes). A dataflow is a series of paths through which an event moves and a specification of the actions that occur within the paths. A path may have distinct stages such as merging several input streams, doing the reconstruction, tagging output results, or creating an analysis tuple. A decision point is a place along a path where code must make some qualitative decision about the event. Whether or not the event will proceed down a path or be steered to other paths is determined by the configuration of a decision point.

The information conveyed by these concepts can be used to build a static *schedule*. The information can be used to constrain an ensemble of module sequences and express how they will work together. Constraint examples can be “path one must come before path two” and “path three only happens if path two succeeds”.

We want to discuss whether it is sufficient to have a single “process” method that handles all types of actions performed on or with an event, or if we should have *several* different methods, each with a different purpose.

4. **CALIBRATION:** We want to make sure we discuss the requirements placed upon the framework and event model by the specialized task of generating and studying calibrations. There are also issues to be discussed in the subject of *using* calibrations.
5. **INTERACTIVE USE:** We would like to discuss the special demands placed upon the system by the need to support interactive use. This includes issues of a diverse nature, such as an interactive help system and interactive analysis.
6. **MULTITHREADING ISSUES:** We would like to discuss the possibility of taking greater advantage of the use of multithreading, for example in the parallel reconstruction of multiple events, and parallelism within the reconstruction of a *single* event. We believe this will become more important in the future, as multiple-processor machines and multiple-core processors become more common.

Given the large amount of ancillary data necessary to process events and the large start-up time, there may be an advantage to starting up a single executable and create an event processing “pipeline.” One image of geometry and calibration data can be established along with one instance of the data handling and persistency code. Independent paths (or subpaths) can be executed in parallel. We believe that such schemes may not require physicist-developed code to be multi-threaded. The EDM and framework will do the synchronization and resource scheduling.

7. **DATA INPUT AND OUTPUT:** We want to discuss the writing of multiple streams of output, merging multiple streams of input, and the usefulness of “tagging” events (and objects within events) in the creation and processing of multiple input and output streams.
8. **UNPACKING:** We believe it is useful for the core event model to provide utilities for the unpacking of “raw” data, in the interest of efficiency and ease-of-use.
9. **DATA VISUALIZATION:** Data visualization requires use of the interactive features of the framework. The tools used for this operation typically complicate the system because they have their own concept of an event loop. The data formats necessary to visualize the data can be quite different than those in an event. We would like to discuss how the interactive features of a framework allow for interfacing to visualization packages.
10. **HISTOGRAM AND NTUPLE SERVICE:** We recognize that ROOT will be used by many, if not all, CMS collaborators. We think it is likely that one of the common uses of the analysis framework will be to produce programs that make ROOT ntuples (or TTrees) specialized for a specific task. We also expect there to be a need for the use of histograms during reconstruction especially during the commissioning of the CMS detector. We believe that the mechanisms provided by ROOT for the management of histograms and ntuples are inadequate for serious use, and that a *histogram service* should be provided to handle the management.
11. **SPECIFYING INPUTS AND OUTPUTS:** Is it a good idea for modules to declare a list of necessary inputs and products? If so, how is this expressed? It is likely that object types is not good enough and that some of the EDM metadata is necessary (*e.g.*, algorithm name and version or configuration parameter values). How is the static declaration part expressed? If there is a dynamic component (one that uses metadata configuration), when does it need to be calculated? Is it before or after module construction time?

Is this information and the dataflows and decision points information enough for the framework to produce a deterministic schedule? We desire a schedule that eliminates redundant work and minimizes the chances of inconsistent results due to order dependencies within a program configuration.

12. **ERROR HANDLING:** We believe that the algorithms contributed by physicists and others that plug into the framework cannot determine directly know what actions are required as a result of an observed adverse condition. The context in which the program is running will dictate what the proper actions will be when these conditions occur. This implies that modules and algorithms report conditions and the framework determines the proper course of action<sup>3</sup> (e.g., abort event processing, ignore, skip a portion of processing, abort a run, save the event for further testing, restart the program). We would like to discuss how these conditions are reported by user code and how the framework makes use of the information.
13. **CALIBRATION AND ALIGNMENT DATA MANAGEMENT:** We believe it is important for it to be easy for users to obtain the appropriate calibration data and alignment data for a given data sample. We think it would be useful to discuss how the framework can automate this process.

## 4 A Comparison Between CDF and CMS

In order to facilitate discussion during Vincenzo's visit to Fermilab, we present a description of the CDF architecture and a comparison with *our understanding* of the CMS architecture. We have two goals:

- To present for Vincenzo the context from which those with CDF experience approach the framework and event model, and
- To express our understanding of the CMS infrastructure, so that any misunderstandings we have can be corrected.

### 4.1 The Architecture View

#### 4.1.1 The Module Layer

A good model for CDF's framework is a software bus or backplane, which provides a well-defined common interface for a set of interchangeable components.

---

<sup>3</sup>The framework should allow run-time configuration to determine the course of action to be taken when a module fails.

The components in this system are *modules*, which are the highest-level elements of the architecture. Modules can be plugged into this bus as long as their event input requirements are satisfied by an earlier module in the chain or by the input data module. Modules can only communicate through the event record, which is described below.

An executable is composed of the framework, plus an arbitrary number of different modules that are specified at build time. At run time, the user can choose to run any subset of those modules in any order (although obviously not every order would be sensible). The module concept is important because it enables unit testing; each module can in principle be run independently of any other module, as long as the proper set of inputs exist in the event record. In-between module invocations the framework can provide optional services such as checking memory usage; this allows leaks to be isolated to particular parts of the binary..

Modules are encouraged to be single purpose, dealing with a limited set of inputs and outputs (although “limited” does not mean exactly one). This does mean that there can be a large number of modules associated with a particular task, such as reconstructing muon hits, stubs and track-stub combinations throughout the detector. These modules must be sequenced in just the right order. For that purpose, we have the concept of a *sequence* of modules, which is an ordered list of modules.

Modules also provide the run-time configuration interface. The current state of all of the modules parameters can always be “shown” and a description of the meaning of a parameter can always be printed through a generic help interface available through run-time commands. <sup>4</sup>

There is a special class of modules that provide services such as:

- Event Input
- Multiple Stream Output
- Geometry Input
- Calibration Management
- Run Configuration Input
- Error Logger Management

---

<sup>4</sup>CDF has the ability to script configurations in tcl. Most CDF users would say this is a good thing. Our run job collaborators hate this, so we think this is one area where it impossible to come to a Run II experience consensus.

CDF modules have a minimal state machine concept. Module writers must think about what actions they want to do for specific states of the process. The most common states are:

- Module Construction
- Begin of Job—happens after all user configurations
- Begin of Run
- Event Processing
- End of Job
- Module Destructor

Typically, at begin of job, the module performs initializations that are needed for later event processing. At begin of run, the module will read database information that describes the detector state for a particular run. In the event-processing section of a module, the framework provides all of the data associated with a particular beam crossing, and the module typically makes use of some subset of that data for reconstruction and analysis purposes. There are more possibilities (*e.g.*, `beginFile`), but they are not in common use.

Modules are the creator of event objects, which are described in greater detail below. They create the objects on the heap and assign EDM (event-data model) handles to them. (The handle is created on the stack of the module's event-processing method). The EDM then takes care of their lifetime management, but it is up to the module to decide whether or not the object it creates should be appended to the event. Only appended objects are visible to downstream modules; otherwise they are destroyed when the module has completed its processing. (Technical details: If the object is appended to the event then its reference count is incremented. When the handle goes out of scope and is destroyed it decrements the count associated with the event object. If the object was never appended to the event then the reference count will go to zero and the object will be destroyed but if it was appended the count will be at least one and the object will live on until the event gets cleared. This happens right before the next event is read in.)

Module packages are not allowed to depend on each other. If a module wants to use a method of another module then it must define an algorithm class that encapsulates that functionality that then can be shared between modules. From a user's perspective, he can then prepare an unordered list of module packages he wants linked into his application, and the system can then calculate which libraries to link.

CMS: No clearly identifiable set of single purpose modules, with clearly defined precursors.

### 4.1.2 The Algorithm Layer

As was just described, classes in the algorithm layer are those that might be used by multiple modules. Ideally, algorithms use event data that is passed to them through a shared pointer, and then report output and errors back up to the modules that invoke them. They depend on modules to configure services such as calibration database access. If an algorithm is in fact used only by a single module, then this layer is unnecessary, but as it can be difficult to predict how users may want to use particular algorithms in the long future of the experiment, it is safest to define classes in this layer. Since this layer is designed for reuse, it is natural that the different algorithm libraries depend on each other. However these dependencies must be kept in one direction. There can be no cyclic dependencies. CDF has found that the best way to assure this is to align the ordering of the algorithm libraries to the ordering of the object libraries. For example a jet algorithm may use a function from the calorimeter library but not vice-versa.

### 4.1.3 The Objects Layer

Algorithms ultimately read and then create data objects. The classes for these objects must be partitioned into their own packages and libraries. The most important feature of the data organization in CDF is that the data classes themselves are as simple as possible. For the most part, the objects are structs of data and member functions that manipulate that data to obtain derived quantities that no one would ever argue about. For example if a four-vector is stored there may be a member function that would return its invariant mass because no one will ever say that they have a better way of calculating that value. On the other hand, a member function that calculates a parameter characterizing the longitudinal shower shape of an electron in the electromagnetic calorimeter is an example of a function that *never* should be in an Objects level package. Schema evolution is very painful (it makes older validated versions of the code unusable with newly produced data; you can't expect forward compatibility) and should be avoided at all costs. We need to know that when an Object level package changes it is not due to some algorithmic change but rather a change to the layout of the data.

Ideally, object packages would only depend on infrastructure classes like `StorableObject` (similar to `TObject`). It is often convenient to use geometry information to organize the data. However it should always be possible to read and

interpret a data object without knowing which version of the geometry was used to organize it. CDF made the mistake of not following this rule in the silicon detector system and it was one of the things we had to correct later. At CDF we now use the following guideline: a compile time dependency on a geometry package is allowed if the contents of that header file will never change. For example, you can use the muon wedge numbering convention defined in a header file, but cannot refer to the exact spatial position of the wedges or their wires.

In summary, classes in the object layer may depend on the static pieces in the geometry layer as well as on infrastructure and utility layers. They can not depend on any objects in the algorithm or module layers.

CMS has no clean separation between the Object Layer and the Module Layer.

CMS has no enforced restrictions on what may be contained in an Object Layer class.

CMS persistent objects:

1. may contain simple data (doubles, ints, etc.)

*OK*

2. may contain persistent POOL references (using global OIDs) to other objects.

*This is not desirable. Object IDs need to be unique only within the event. Globally unique IDs allow objects within an event to refer to objects within a different event, which is not desirable. Also, the ID of an identical copy of an object should be the same for every copy. However, POOL assigns a new ID to each new copy. Furthermore, POOL does not assign the ID until the object is marked for persistency, so newly created objects have no ID. Furthermore, POOL persistent pointers are not simple objects, so they complicate the browsability of data. The conclusion is that event objects should not contain POOL persistent references. We will refer to this conclusion as Rule A.*

3. may contain C++ pointers to other objects, sometimes embedded in a "smart pointer" class.

*This is probably not desirable. Objects referred to by C++ pointers are embedded in the referring object by ROOT. However, the format in which the embedded object is stored*

*is more complex than if the object were contained directly. This may complicate the interpretation of the event object by tools other than POOL or ROOT. The conclusion is that event objects should not contain non-transient C++ pointers. We will refer to this conclusion as Rule B.*

4. may contain arbitrarily complex object members.

*This is not desirable. Use Event local IDs as references to keep the individual objects simple. This is Rule C.*

5. may contain `vectors` or other containers of 1–4.

*OK for 1.*

6. may contain arbitrarily complex methods (i.e., member functions).

*This is not desirable An object should contain only methods that never change. This is Rule D.*

7. often inherit from one or two arbitrarily complex base classes.

*This is not desirable. Inheritance from base classes containing no data (only methods) is acceptable if the methods will never change. Inheritance from base classes containing arbitrary data is undesirable, as this complicates the interpretation of the object. This is Rule E. Inheritance from a base class specific to the persistency mechanism is acceptable.*

Here is an analysis of the classes used for persistent objects to store an event in CMS data sets. Only per event objects are described here. Metadata are not discussed.

For the `SimHits` per event persistent objects:

1. 4 COBRA classes

(`GenEventBody`, `SimEventBody`, `PythiaSimEvent`, `RawEvent`)

`PythiaSimEvent` is complex, having inheritance and POOL pointers to `GenEventBody`, `SimEventBody`, and `RawEvent`. Violates Rule A and possibly Rule E.

`RawEvent` is complex, having `vector` of POOL pointers to a whole mess of stuff. Violates Rule A.

`GenEventBody` and `SimEventBody` have data members of class type. The data members are simple, although they have methods.

2. Wrapped STL vectors (some with external hashing) of 2 COBRA (Profound) classes:

(PCaloHit, PSimHit)

PCaloHit is simple.

PSimHit is simple, except that it contains member objects of type 3-vector and 3-point from the COBRA “class reuse” library. The data here is trivial (e.g., 3 floats), but the classes have many methods and levels of inheritance.

*Summary:*

*PythiaSimEvent and RawEvent are too complex, violating Rule A, and possibly violating Rule E.*

*GenEventBody, SimEventBody, PSimHit, and PCaloHit are OK.*

The per event objects in a Digis data set are instances of:

1. 3 COBRA classes. All 3 are complicated.

(PRecEventInd, SimCrossing, SimDigiEvent) (inheritance, POOL pointers, etc. Violates Rule A and possibly Rule E.)

2. Wrapped STL vectors (some with external hashing) of 12 ORCA classes.

Of the 12 ORCA classes: None have member functions that seem likely to change.

Three are simple (ints, floats, doubles, etc. only). GOOD. (MRpcDigi, MuEndStripDigi, MuEndWireDigi)

Four use member structs (with or without bitfields) for packing, but are otherwise simple. Member structs have no methods. (Digi-SimLink, MuBarDigi, PixelDigi, StripDigi)

Two have a single data member of class type. (Violates Rule C). But the data member is simple, although it has methods. (Calo-DataFrame, EcalSelectiveReadoutTower) contain a PCellID.

Two have a single data member of class type. as above, but also inherit from a base class. (Violates Rule C and Rule E) The base class has no data, only methods. (EcalTrigPrim, HcalTrigPrim) contain a PCellID and inherit from CaloTriggerPrimitive (abstract). CaloTriggerPrimitive has methods only (all simple), and does not inherit.

One has several data members of class type. Violates Rule C. (L1TriggerReadoutRecord) The types of the data members have not yet been checked.

*Summary:*

*PRecEventInd, SimCrossing, and SimDigiEvent are too complicated, violating Rule A and possibly Rule E.*

*L1TriggerReadoutRecord may be OK.*

*CaloDataFrame, EcalSelectiveReadoutTower, EcalTrigPrim, HcalTrigPrim, MRpcDigi, MuEndStripDigi, MuEndWireDigi, DigiSimLink, MuBarDigi, PixelDigi, and StripDigi are OK.*

For the DST, all reconstructed objects inherit from RecObj, violating Rule B, and the RecObj pointer is packaged in a PRecCont.

1. PRecCont is an STL vector of own\_ptr<RecObj>. It has no base classes, and no other data members.
2. own\_ptr<T> is COBRA's own smart pointer. It has no base classes, but many methods. Its only data member is a T\*, or in this case, a RecObj\* (Violates Rule B) . Any pointed to object of a persistent capable class inheriting from RecObj will be output in its entirety. This is due to a feature of ROOT (transparent to POOL).
3. An analysis was done of the reconstructed objects used for tracks in the silicon tracker. Many of these classes have multiple layers of inheritance, and many methods at each level. Many of these methods are simple and should never change. However, this is probably not the case for all such methods. The data members, however, are all simple floats, ints etc., and STL vectors of such.
4. No analysis was done at this stage about the complexity of the remainder of reconstructed objects (i.e., other than the tracker).

*Summary:*

*Reconstructed objects are too complicated. Although their data are simple, there are many methods, and many levels of nested inheritance (Violates Rules D and E). Also, the implementation of RecObj through pointers may cause*

*difficulties (Rule B). These topics need more investigation*

#### **4.1.4 The Geometry Layer**

Static geometry information such as positions of tower boundaries in the calorimeter is contained in header files. The same geometry description classes are used for both the simulation and the reconstruction; this design feature allows us to avoid the labor-intensive task of describing the same geometry multiple times for multiple purposes. The geometry classes contain information needed to initialize GEANT 3 data structures as well as interfaces for tasks such as alignment and dead channel marking. These dynamic aspects of the geometry must be kept in databases.

#### **4.1.5 The Infrastructure and Services Layer**

These live below the geometry layer. For the most part it is obvious what goes here; the classes that describe the framework itself, and services such as database access and error logging. The only tricky part is working out the interface between the geometry layer and the alignment DB, which is part of the calibration service layer.

#### **4.1.6 Utilities and 3rd Party Packages**

These are the tools that the above layers all use. By definition these packages will not depend on any experiment-authored software. It includes everything from CLHEP to ORACLE client libraries to ROOT libraries.

#### **4.1.7 PackageList**

This is the utility that has kept link time problems off the CDF help lists. It provides the ordered list of libraries for all CDF packages while minimizing the exact set that is used for any one job.

**CMS:** There is no simple way to determine the minimal (or even a sufficient) set of libraries needed to run a specific job, other than trial and error. In order to run our sample Si tracker use case, no fewer than 75 COBRA or ORCA libraries needed to be specified at link time in the SCRAM build file. If any subset of these 75 libraries was not included

in the SCRAM build file, any one of the following results could (and did) occur:

- A failure at link time.
- A descriptive fatal error message about a missing library.
- A totally confusing fatal error message.
- An uncaught exception.
- A segmentation violation.

Only five needed COBRA or ORCA libraries, and their six dependencies, could successfully be left unspecified at link time and be explicitly loaded at run time by `dlopen`. These five libraries were selected in no fewer than three different ways in COBRA. If CMS has a general mechanism for run-time loading of libraries, it is not widely used.

## 4.2 EDM

### 4.2.1 The Event

- The EDM provides a well defined *event* concept, with well defined interfaces to the data within the event.
- The EDM can handle data with different lifetimes or periods of validity. For instance event data is destroyed after every event, but run dependent data would be preserved for the lifetime of the run.
- The event must provide type safe interface to objects stored in the event.
- The EDM allows a separation between transient and persistent representations of the data. The most important feature of the persistent data is that it be as compressed as possible with minimal information loss. The most important feature of the transient data is ease of use and navigation. This argues that there should be some non-trivial transformation of the data between disk and memory. By default a minimal amount of the event should be transformed this way. The rest should be turned on by the the user (or transformed on demand, which CDF does not do).
- It has an interface to the configuration system, so that file level metadata may be made persistent in an optimal way. The EDM together with the framework should assure that object provenance is completely retrievable. At CDF all of

the parameter settings of the module that created the object are summarized by an `RCPID`. The `RCPID` is stored as a data member of a `StorableObject`. The `RCPID` can be used as a key in a database lookup to retrieve the values of the parameters associated with it.

- At CDF, the event consists of an expandable collection of data objects, which are described by classes in the object layer described above. Once an object is appended to an event it becomes read-only. The non-`const` version of the handle is emptied of information and a `const` handle is passed back to the caller of `append` method.
- As event objects are read-only, a strategy for correction and association objects should be designed, to handle the fact that information in the object may need to be “improved” later due to greater understanding of the detector. Without such a strategy, users will want to violate the read-only policy and figure out ways around it (usually involving global variables). At CDF we didn’t think about this ahead of time, and we now have several solutions to the problem (but at least we’ve stamped out most of the globals).

#### Current CMS *event* characteristics:

1. No clearly-defined, fully self-contained event class that contains no pointers. Navigation required between the components of an event. This usually requires pointers of some kind. Objects have no unique per-event ID.
2. Lifetime control—An object remains locked in memory as long as a `POOL` smart pointer points to it. Effectively, these are shared pointers. When an object is committed to disk, the shared pointer used for writing the object is destroyed at the commit, so the object will be locked in memory only if other shared pointers reference it.
3. Type Safe Interface—In some cases, a retrieved object may be accessed through a pointer that is not of the same type, but rather is of type pointer to a base class. This does not usually require the user to cast the pointer, as virtual methods can be used to ensure type safety in method invocation. This occurs because of the deliberate use of polymorphic pointers in the framework, not because of any fundamental problem with `ROOT` or `POOL`.
4. No general mechanism to provide different transient and persistent representations of an object, except for the ability to mark some data members as transient.

5. Interface to configuration system—Configuration information is stored in the metadata, but it is not clear if this information is sufficiently complete. Information on the Geometry (when necessary) and the detector components is stored, but it is far from clear if sufficient information to reproduce the results are stored.
6. Per-event data is read only once written, but there is no mechanism enforcing this, other than the framework not supporting it. No mechanism enforces the constancy of transient versions of read-only objects. “Globals” are generally not used.

#### 4.2.2 Collections, Objects and the Associations Between Them

- The EDM should provide a well defined interface for creating associations between collections of objects. At CDF this is provided through the *link* classes. The type of the link class depends on the type of collection it is pointing into. For instance if the link needs to point into a collection object then it must know the link to the particular collection plus the index into that collection.
- It should enforce clear policies of object ownership and provide tools for tracking down mistakes where it is not possible to prevent them by construction. At CDF event objects are supposed to be created on the heap and passed to the constructor of an EDM handle. The handle manages the lifetime of these objects by reference counting them. There are debugging tools for tracing the reference counts of objects that are suspected of leaking out.
- CDF’s EDM provides several collection classes that are light extensions to the STL *vector*. Different collections are used for different ownership policies. There are *value collections*, which own their elements, but there are also *view collections*, which are just collections of references to other objects stored in value collections. This is how we resolve who should write out the data when many people are referencing it. The collections write out their contents, the views just write out their persistent references to them.
  1. CMS has no general purpose interface for providing associations collections of objects. There do exist some specific “link” classes for associations between objects of specific types.
  2. Object ownership—POOL has the responsibility for memory management of objects in its data cache. There is no well-defined

policy for the memory management of transient objects, or transient copies of persistent objects.

3. Collections: COBRA has its own collection mechanism, but it is currently used only for collections of events (a run), and for collections of runs. This is used only internally by COBRA, and is complex (*i.e.*, not light). POOL provides a (not light) collection facility, but it is not yet used. STL `vectors` are extensively used for collections, and are used in the implementation of COBRA's collection mechanism.

## 5 Metadata Management

In the present section, we will refer to the term *metadata* repeatedly. This is an overloaded term that can refer to the object level metadata that an experiment specific application framework (such as ORCA /COBRA) requires to reconstruct complete events from pieces stored in possibly many files, to dataset level metadata that may contain run conditions or Monte Carlo generation parameters, or to event level metadata that may consist of analysis defined quantities of interest to the analyst. The intended usage of the term metadata will be explicitly determined in the following text unless it is completely clear from the context.

### 5.1 CDF Use of SAM

CDF has moved to using version 6 of the SAM metadata. SAM is documented extensively elsewhere, but a short summary follows.

In the SAM system, events are located in file containers with sequential access. The event data model must therefore support at least this kind of file based access to events in order to use SAM. Conversely, the use of SAM is very widely applicable. All kinds of metadata in SAM are generally file oriented. Files are imagined to exist in a multi dimensional space specified by their metadata. A metadata query service interprets user given constraints on these metadata dimensions in order to describe a portion of metadata space containing files of interest. The Sam schema explicitly defines some physics metadata, such as luminosity information. The schema then ties this luminosity data to the runs and data files. Sam also keeps track of some data stream and trigger information. However, most of the streaming and trigger data is interpreted directly by physics applications. Sam is capable of recording the physical data stream, the trigger stream and the trigger lists. (CDF currently encodes trigger information in the "file family" parameter of the Enstore

mass storage system and it's equivalent in SAM.) Other metadata include system level metadata consisting of authentication, authorization, and fabric definition data; production job request metadata, and user defined physics metadata. A pilot project to use SAM with POOL based COBRA/ORCA datasets has met with preliminary success in that it has been shown that the ROOT-based files can be stored into a SAM station and the information needed to reconstruct a POOL XML file can be stored in the SAM schema.

## 5.2 Object-level Metadata and CMS

The current procedure for building a dataset and preparing it for access by physicists using the ORCA /COBRA application has been well-documented elsewhere.<sup>5</sup> The procedure is a fairly involved and time consuming task comprising mainly of manipulation of the object level metadata and metadata contained in the POOL persistency layer. While we understand that some of this is due to issues that came up in the haste to prepare for the 2004 Data Challenge or to issues in the POOL software implementation itself, and perhaps are already being addressed, it is nonetheless important to try and determine as far as possible requirements on the event data model that may ameliorate the situation.

One issue is that the event data model must contain some mechanism to guarantee safety of concurrent accesses to the event data. Previously, this was guaranteed by the OBJECTIVITY lock manager. Without this capability, metadata must be built sequentially (*i.e.*, data segments must be “attached”) in a dataset in order to prepare it for access by physicists. Another issue is that there is no API for the ORCA /COBRA application to tell POOL to close a file or to specify a maximum number of simultaneously open files. The resulting situation is that during a typical dataset build, all of the files comprising a dataset will be open contemporaneously. For a large dataset consisting of thousands of files this will break even the most robust mass storage system access methods, requiring that the entire dataset be staged to a specially tuned NFS partition. (Later versions of POOL support a parameter to specify the maximum number of files open under POOL.)

## 5.3 Dataset-level Metadata and CMS

Dataset-level metadata includes run and trigger conditions collected during real data taking and generation or simulation parameters for Monte Carlo datasets. Requirements coming from this level of metadata are more on the data management

---

<sup>5</sup>See <http://home.fnal.gov/~gug/cms/datasetBuildAndValidate.html>.

system than on the event data model, and thus beyond the scope of the current paper.

## 5.4 Event-level Metadata and CMS

Several groups have experimented with event level metadata, or *tag databases* in CMS back when OBJECTIVITY was in use. Since the event data was directly stored in an object database, the conceptual procedure for creating a tag database was a simple one. One simply created an object containing the tag or other event level calculation and stored it into the event itself. See for example presentations by Koen Holtmann<sup>6</sup> and Vincenzo Innocente *et al.*<sup>7</sup> for CMS, or David Malon and Kristo Karr<sup>8</sup> for ATLAS

In CMS, in order to provide similar functionality again, some kind of indexed or pseudo-random access to events data should be supported by the event data model. In CMS, this is possible once the dataset has been fixed. This is also possible in CDF, where a tag database has been implemented using root trees containing run numbers, event numbers, and file pointers.

## 6 Summary

We recognize this document is of significant size and scope. We collect here what we believe to be the most important issues we have addressed. In most cases, we suggest a goal, but we present neither concrete designs nor implementations to achieve these goals, which are:

1. an explicit `Event` class, as defined in §3.1.
2. explicit framework modules, as defined in §3.1. These modules should communicate with each other only *via* the `Event`.
3. plans for the designing (when desirable) of improved data objects, in which each data object contains only simple data and is isolated from the algorithms that creates it.
4. a system for scheduling the execution of modules (§3.6) that is efficient, deterministic, and easy to use.

---

<sup>6</sup>See <http://www.ppdg.net/pa/ppdg-pa/idat/caltech-may02/koen2.ppt>.

<sup>7</sup>See <http://www.ihep.ac.cn/~chep01/paper/3-041.pdf>.

<sup>8</sup>See <http://agenda.cern.ch/askArchive.php?base=agenda&categ=a043071&id=a043071s1t2%2Ftransparencies%2FMalonEventTags.ppt>.

5. an association of event objects with appropriate metadata to allow querying through the event interface.
6. a simplification of physical design to remove unnecessary library dependencies.
7. a more robust and user-friendly system for the explicit<sup>9</sup> loading of dynamic libraries.
8. a long-term plan to gain benefits from the use of multithreading (§3.8, item 6), or to be rid of the additional complexity multithreading introduces.

We look forward to collaborating with you in addressing these issues.

---

<sup>9</sup>Loading with `dlopen` on \*nix.