The CLEO III Data Access Framework

Martin Lohner, Christopher D. Jones, Simon Patton, Michael Athanas, Paul Avery

University of Florida, Gainesville

Abstract. The CLEO III Data Access Framework allows users to access data from various data Sources (files, databases, etc.) in a uniform manner, to process the data, and write out new data to Sinks. The central access point to data in the system is the *Frame* which allows uniform and transparent access to all data, including event data and constants. This represents a break from event-centric models. Algorithms can be written in a time-ordered or in a lazy-evaluation fashion. This paper describes in detail this data access framework, which we think is more flexible and powerful than the standard event-centric frameworks.

INTRODUCTION

The CLEO III experiment will collect on the order of 100 TB of data during the first few years of its lifetime. The challenges facing CLEO III are how to analyze such a large dataset efficiently and flexibly while providing a smooth transition from the current CLEO II environment to the new environment.

The design of the new data model benefits from the history of the CLEO II data model, both in its short-comings and successes. Having to learn different analysis frameworks, one for reconstruction, another for fast analysis, yet another for calibration has proven to be a major obstacle to most physicists on the CLEO II experiment. The APIs for data access in the different frameworks are directly connected to the data formats, but are specialized and well-tuned for their tasks.

The new CLEO III framework allows reconstruction, analysis, and calibration in one system. Furthermore, in contrast to event-centric models, all data (events, beginruns, calibration, etc.) are treated equally. The system is extensible, permitting users to incorporate their own data and allows for inevitable changes in data formats.

We support both C++ and Fortran for coding. Although users are encouraged to start new projects in C++, we are supporting vast amounts of legacy CLEO-II code written in Fortran.

We have made this framework available to CLEO physicists, a small but steadily increasing number of whom are using it for CLEO II data analysis. We present

here a detailed design of this framework.

THE FRAME MODEL

When a physicist wants to process CLEO data, he is interested in the state of part, or all, of CLEO at one or more instances in time. Some parts of that state are fixed over a long time, while others have a short life time. We bundle data related in lifetime and concept together in *Records*. Some examples of a Record are an entire event; subdetector geometry information; and beginrun data.

Records of a given type are organized into a *Stream*, where each Record represents a change in the state of CLEO. Our data are treated as a collection of these time-ordered Streams, as shown in Fig. 1. A snapshot of the state of the detector in terms of the relevant Records pertinent to a specific time is called a *Frame*.

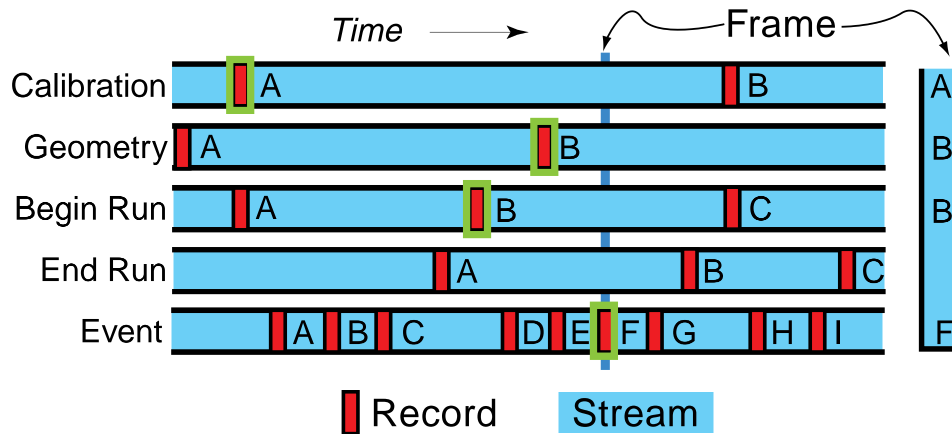

FIGURE 1. Graphical representation of the Frame model.

This model has several nice features. First, it mimics data taking; a physicist can easily understand how it works. Second, it is straightforward to add user data to a Stream or even add new Streams. Finally, since all Records are treated equally, one can study events as easily as correlations between other kinds of data, e.g. between noise and beam crossings.

PROCESSORS

A user usually wants to run an algorithm whenever a particular type of Record appears on a Stream (e.g. a new event, or a new hardware Record). This behavior is made possible by a *Processor*, which is simply a *container of algorithms*, each of which is bound to a Stream. By binding an algorithm to a Stream, the algorithm is triggered automatically by a new Record appearing on that Stream.

The result of an algorithm may be stored in the Frame, thus the order in which Processors are executed is important. If Processor B needs the results of Processor

A, then Processor A needs to be run before Processor B. This dependency leads to the notion of a *path*, which is an ordered sequence of Processors, as shown in Fig. 2. All the algorithms bound to the same Stream are executed in sequence in

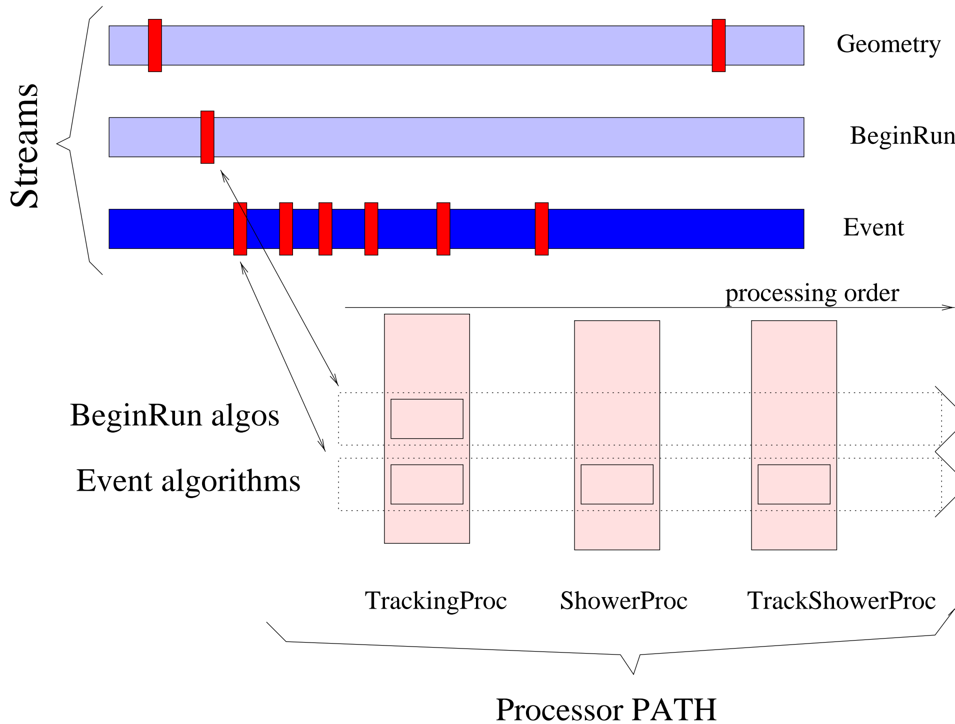

FIGURE 2. A Path of Processors showing the order in which they are executed.

the order in which the Processors were selected every time a new Record appears in that Stream.

PROXIES

In the user's mental model the data sit in the Records in the Frame, ready to be retrieved. It is a simple model. In reality the data in these Records are managed by objects called *Proxies* [1]. Upon request the Proxy returns the item, or, if the item does not yet exist, the Proxy creates it. The Proxy caches the information for future inquiries, flushing the cache when a new Record appears.

The Records represent type-safe containers of data, i.e. asking for a track will trigger the *TrackProxy* to return a track. The type-safety is achieved via a unique key based on the type of the item for type-safe retrieval. Other types of keys are used to further tag the data.

What is the advantage of using Proxies? For one, the data objects are independent of the algorithms that produce them, simplifying their design (see Fig. 3). For instance, if the user wants to design a Track class he does not need to know

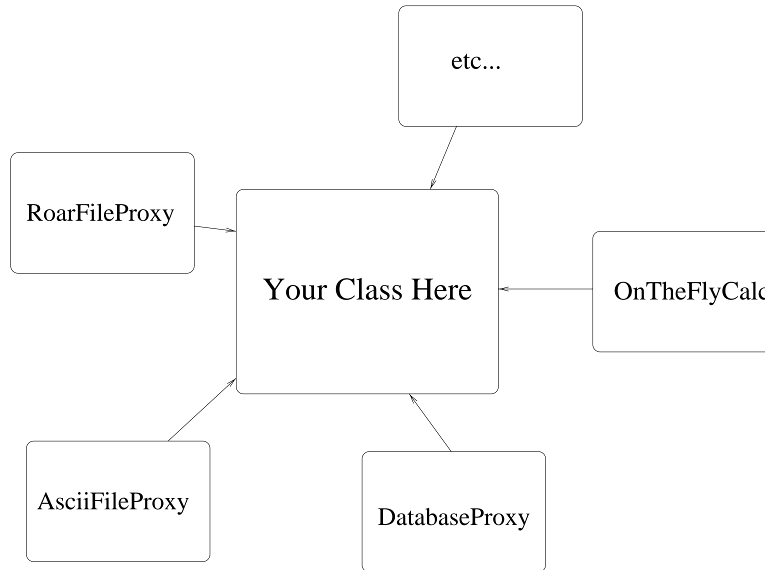

FIGURE 3. An Object and the Proxies that can create it. (Only one Proxy may create the object in a given job.)

how a *TrackFitter* will create the Track. Furthermore, since the Record is merely a container, adding new types of data (and Proxies) is easy. Another advantage is *lazy-evaluation*; no Proxy is called upon to execute its algorithm and create an object unless the object is asked for.

Proxies have access to all information in the Frame; a Proxy may ask other Proxies to supply data it needs to perform its calculation. The execution order is programmed by the Proxy designer: if Proxy B needs the result of Proxy A to perform its duty, it will trigger Proxy A to supply that data, then continue its calculation. This built-in execution order differs from the runtime-selectable Processor execution model discussed above.

To illustrate, let us use a potential algorithm to find track-shower matches. The Proxy in charge of track-shower matches will first request a list of tracks from TrackProxy and a list of showers from ShowerProxy. If this is the first request to those Proxies, they will create tracks (and showers) by reading them from the database or by running a track finding and fitting algorithm. The track-shower Proxy will execute an algorithm to match tracks to showers and return the resulting matches.

Proxies are natural for creating *composite* objects. Such objects consist of, or are built on top of, a number of smaller objects. Instead of creating the entire composite object at once when requested, only the parts needed are created, the ultimate lazy evaluation scheme. Another type of macro object is the *navigational object*, which provides a common access point to related information, e.g. all the various pieces of information related to tracking.

For easy management, we group Proxies which are related in purpose or func-

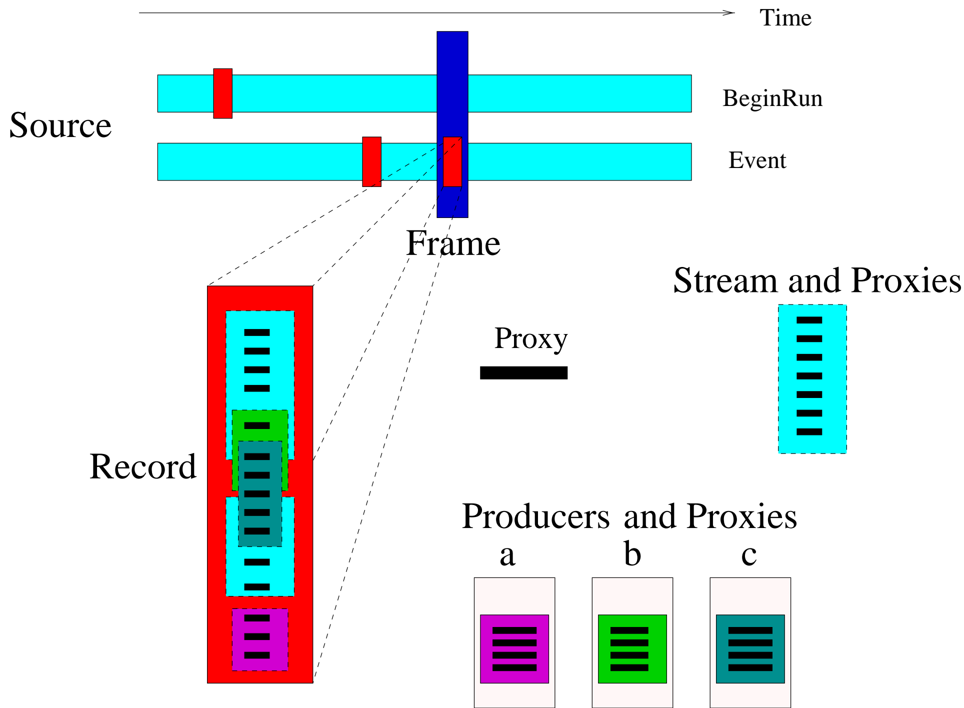

FIGURE 4. Proxies.

tionality in *Producers*, as shown in Fig 4.

SOURCES AND SINKS

We can also employ Proxies to retrieve previously saved information from a *Source* (e.g. a file or database). This type of Proxy simply retrieves the proper information from such a persistent store and recreates the object in memory. All the Proxies for a Source make up a *Sourceformat*. There may be a Sourceformat for reading objects from a file, and another (producing the same objects) from a database. Similarly we employ a *Sinkformat* with Proxies to write objects to a *Sink* (file or database etc.).

Employing Proxies makes it easy to support multiple formats. Data can be delivered from and stored to any number of CLEO-specific legacy formats and new formats including a database. Multiple Sourceformats and Sources can be used at the same time and can be specified at runtime. The same applies to Sinks.

Proxies from Producers may override Source Proxies. For example, a Sourceformat may contain a *TrackProxy*, retrieving tracks from a database. If a Producer with another TrackProxy is selected, asking for tracks will trigger the tracks to be refit, overriding the Source's Proxy.

USER INTERFACE

With the data sitting in the Frame, ready to be retrieved, accessing the data means merely "extracting" the data from the proper Record. The call to the Proxies is hidden behind smart pointers, *FAItem* and *FATable*, for singly- and multiply-occurring items respectively:

```
FAItem< BeamEnergy > beamEnergy;   // single item
extract( beginrunRecord, beamEnergy );

FATable< Track > tracks;           // multiple items
extract( eventRecord tracks, ‘‘good’’, ‘‘MyTracker’’ );
```

Note that the method for retrieving data from a Record is the same for all Records.

The extra parameters in these extraction calls are used to distinguish between different *uses* and *productions*. For instance, a track can be categorized as being of "good" or "bad" quality, and having been produced by a particular Tracking Algorithm *MyTracker*. If the production label is not specified, a default value is used.

SUMMARY

We have presented the new framework for CLEO III data access. This system is based upon the Frame, which represents the state of the detector at a particular instant in time and provides a uniform access point to all data. The framework is data-centric and acts as if all data are available, and the user merely has to ask for the data.

We favor our data-centric model over a component model. In a component model (e.g. CORBA or COM), a user asks for an interface to create an object. (For instance, ask a *Tracker* to create tracks.) In a component model a Source would have to have many different interfaces to be able to create all the different types of data. In our design the user asks for the data directly. We believe our data-centric model is much easier to use and provides the same flexibility.

We present the user interface and job control for this framework in a separate submission to this conference [2].

This work is funded by the Department of Energy, grant DE-FG02-97ER41029.

REFERENCES

1. Gamma *et.al.*, *Design Patterns*, Addison Wesley 1995.
2. M. Lohner, C. Jones, P. Avery, *Suez: Job Control and User Interface for CLEO III*, CHEP98 Conference Proceedings.