Software Infrastructure Support for Analysis

Marc Paterno
Jim Kowalkowski
FNAL/CD/CEPA
CD-doc-435-v0
BTeV-doc-3278-v0

Contents

What is not ordered to an end is not an effect.
— JOHN DUNS SCOTUS

1 Introduction

1.1 Motivation

CDF and DØ, the Run II experiments at Fermilab, provide the best laboratory for the study of modern software practices in the context of the analysis of hadron collider physics data. Experiments planned for the near future (within a few years of the time of this writing)[1] may benefit from analysis of these experiments' experience in the development of software for data analysis. We have written this document to collect and study the experience of CDF and DØ, concentrating on subjects most relevant to the software written to support physics analysis by physicists.

1.2 Scope and Coverage

We will primarily address *final analysis*, including the (often multi-step) transition from the output of batch reconstruction to data formats used for physics analysis. We will tangentially address *reconstruction*, as the core infrastructure software usually satisfies the needs of batch reconstruction of data.

[1] July 2004.

To support final analysis, we must consider the various manners in which physicists prefer to work. First, and perhaps most importantly, we recognize that *there is no one solution preferred by all*. Different members of the community have different preferences in and requirements for their analysis tools. While there is some broad commonality, too many of the specifics differ for us to believe that any single solution will satisfy all members of a large collaboration. We believe, therefore, that it is necessary for an experiment to support several mechanisms, to enable each physicist to be most productive by working in the mode he prefers.

1.3 Purpose

To this end, we review the analysis methods in common use at CDF and DØ, describe how data are currently transformed, explain the reasons for the common uses, and discuss what infrastructure software is needed to support such analysis. We pay special attention to those aspects of the systems that are most problematic—and perhaps most contentious—so that some of these problems may be avoided in the future.

2 Reconstruction and Analysis Frameworks

2.1 Intended Role

The CDF and DØ experiments each have a modular reconstruction and analysis framework. These frameworks were designed with the intent of being used in several environments:

- in the online trigger system,
- in the batch reconstruction farms,
- in the batch production of simluated sampples,
- in the batch re-processing ("correction") of reconstructed data, and
- in the individual physicist's analysis of data.

In particular, it was expected that the use and development of the "correction" algorithms would be done in the context of the reconstruction and analysis framework. These corrections were to be done in a module that was allowed to *modify* the event data, while an *analysis* module was allowed to inspect, but not modify, the data.

In addition, it was expected that the physicists would produce analysis-level histograms in the context of the framework. Viewing and fitting of histograms would be performed in an external tool, and when creation of new histograms was needed, another pass through the data would be taken using the framework. Each of the frameworks has "hooks" that were intended for use by analysis modules, and these modules were expected to directly interrogate objects contained in the reconstruction event-data model.

In practice, the frameworks were not used to this extent.

2.2 Actual Role

Relatively few of the CDF and DØ collaborators have used their respective analysis frameworks for direct production of final-analysis level histograms. Many more use their experiment's framework as a tool for creating tuples of their favorite sort; these tuples are then used for the creation of analysis-level histograms, and often for the development and application of "correction" code. A majority of users avoided using the objects (of the event-data model) produced by the batch

processing system. In some cases, professors thought they should "protect their students" from the "complexities of the data" by transforming data from the event-data model into a tuple-like structure. Since both experiments' frameworks were designed with use for analysis in mind, it is interesting to determine why they have not often been used for such.

Various reasons are put forth by the users:

- It is too hard to understand the data in the format of the event-data model; interacting with the data is too complex.
- Building executables is too hard; there are too many steps involved.
- Building executables is too slow.
- The resulting executables are too large; too many libraries are needed.
- The resulting executables are too slow; processing data in the available format is too slow.
- The resulting executables take too long to initialize; the delay before the first event is seen is inconvenient.
- It is too bothersome to write C++ code for simple analysis tasks.
- Handling ROOT histograms from a C++ program is harder than using the ROOT prompt.
- They want to use another environment (for example Python) to work with their data.
- Specifying the configuration and inputs of a framework job is too complex.

Some of these reasons are statements of user preference, and other are purported statement of facts about the reconstruction and analysis core software. Our purpose is not to argue whether such statements are true; the fact that such has been perceived, and such opinions are held, is the point of our observation.

2.3 Summary

Experiments must work to avoid such perceptions where possible, but must also realize that some such perceptions are inevitable. The requirements of an analysis system are different from, and to some degree incompatible with, the requirements of a framework and event-data model used in the trigger and reconstruction systems. CMS must therefore work to have the core infrastructure software support the physicist's wish to work in a software environment that enables each physicist to perform data analysis to the greatest degree possible, given the finite resources available.

A particularly bad effect of the perceived difficulty of working in the framework environment is that fact that algorithmic code ("correction" code) has been introduced in the tuple-based analysis environment. This is bad for several reasons:

- Metadata is not kept to indicate what corrections have been done, or what versions or parameters have been used.
- The order in which corrections are done is often ill-controlled.
- When the order *is* well-controlled, this is because a private "mini-framework" has been introduced.
- Because the corrections are not done in the production environment, the code is often not under revision control, and the processing is not sufficiently well documented.

Becaue of these defects, the correction process is not sufficiently understood or reproducible.

3 Data Formats Used in Analysis

CDF and DØ differ in the format in which the output of batch reconstruction is written to permanent storage. DØ uses EVPACK, a modification of DSPACK; CDF uses ROOT files. Thus it is not possible for DØ physicists to directly read reconstruction output files from a ROOT session[2]. While it is technically possible to read CDF reconstruction output files from a ROOT session, this is not a manner in which any significant number of physicists work.

In both collaborations, physicists transform the data from the format used for batch reconstruction into a format which seems (to the individual performing the analysis) more suitable for use in analysis. In this section, we discuss the data formats most widely used.

3.1 Simple `structs`

Simple `structs`, or their equivalents, stored in a variety of formats, are widely popular. By "simple `struct`" we mean a data format that requires no experiment-specific software for its interpretation. Some such constructs, widely used, are:

- Fortran `common` blocks,
- C `structs`, and
- C++ `classes` with only trivial "getter" and "setter" methods.

These are all similar in that they are *tabular* data, which may be easily stored, retrieved, and manipulated, in a variety of file formats. The formats in use at CDF and DØ include:

- ROOT tuples[3],
- PAW ntuples,
- *Excel* spreadsheets[4],
- *R* dataframes[5], and
- ASCII files.

3.1.1 Benefits

Direct manipulation of event-data objects in the reconstruction output requires understanding many facilities of a complex language (C++). Manipulation of data in the form of simple `structs` requires understanding of far fewer features of the language, (or perhaps none, if direct data inspection tools are used). Thus there is a much smaller learning curve associated with the use of `struct`-like data. Of particular importance is the fact that these facilities may not all be useful for doing analysis.

At DØ, users unfamiliar with C++ templates often found the template mechanism by which access to objects in the event is obtained to be confusing. This mechanism was designed to make such access easy, but because it used a feature of C++ unfamiliar to new users of the language, it was a stumbling block for those new users. At CDF, the complex iteration mechanism used

[2] DØ has available, but does not current use, a ROOT-compatible version of their event-data model.

[3] We use the term ROOT *tuples* to include *TTree* and its subclasses, such a *TNtuple* and *TNtupleD*.

[4] *Excel* is the spreadsheet component of *Microsoft Office*

[5] *R* is the free implementation of the *S* programming language and data analysis environment; see http://www.r-project.org/.

to access objects in the event data was a stumbling block for new users. Being unfamiliar with the common C++ iterator idioms, new users had a difficult time adjusting to the use of iterators rather than direct use of collections.

ROOT's *TBrowser* class, its *TTree* class's "full split" mode (a `struct`-level column-wise ntuple) and the `Draw` language[6] work best with simple `structs`—not objects with private data. This is because private data often has a compact nature, it not labeled for convenient use, and may not be meaningful without the manipulation of the public interface of the class.

Because the `Draw` language is not sufficient for all tasks, ROOT makes available the `CINT` language[7] can be used for more complex tasks. For still more complex tasks, one can write C++ code, which can be compiled and dynamically loaded using `ACLiC` [8] Neither of these is a feature of the `struct`-like data format. Some users view writing their own simple event loop inside of a CINT macro or C++ function to be preferable to use of the event loop provided by the reconstruction and analysis framework.

Typically, no language-specific or experiment-specific code is needed to understand data stored in such a format. An off-the-shelf version of the chosen analysis tool (often ROOT) suffices.

The storage of data in simple `structs` facilitates access to those data from other software systems. Some such systems are: other analysis toolkits (such as Java Analysis Studio (JAS) or R), scripting languages (such as Python or Ruby), relational database management systems (RDBMS, such as Oracle, MySQL, PostgreSQL, and SQLite), and from any software tool with a C-language interface.

3.1.2 Drawbacks

Lack of abstraction ability makes handling of complex tasks more complex—this argument is essentially the argument for object-oriented design. In particular, such things as "correction code" becomes a significant maintenance burden. Algorithmic code that uses these `structs` is often hard to analyze, and thus hard to optimize, if it is slow. Furthermore, such code is sometimes unreliable, because code that is hard to analyze is hard to test, and to prove correct.

During the CMS DST 2004 workshop, opponents of `struct`-like formats noted the lack of metadata. Much of the use of this data format by CDF and DØ is consistent with this observation. This lack of metadata is not an inherent feature of `struct`-like data; it is a feature of such data being an afterthought, without sufficient design.

3.2 Object Instances in ROOT Files

Object-like data consists of instances of non-trivial classes. In general, these contain private data, which are in a format convenient for implementation of the object, and which are often *not* convenient for direct manipulation. Such objects are intended to be used through an interface, which manipulates the private data. This manipulation can include non-trivial transformations, and may allow for multiple views of the data. language-specific navigation methods may be presented to associate related objects.

---

[6] The `Draw` language is the special language implemented by the *TTree* class's `Draw` function. The ROOT documentation does not name this language; we name it `Draw` for reference in this document.

[7] The `CINT` language is the language implemented by the CINT interpreter embedded in ROOT. This language is related to, but differs in significant ways from, the ISO C++ programming language. The ROOT documentation often refers to this language as C++, but we find it convenient to distinguish between the two.

[8] `ACLiC` is ROOT's "Automatic Compiler of Libraries for CINT," which uses the same C++ compiler used to build ROOT to build dynamic link libraries containing arbitrary C++ code which may then be called from the ROOT prompt.

In contrast to `struct`-like data, which can be stored in a variety of formats and has been used with a variety of tools, ROOT is the only object persistency mechanism in wide use in the HEP community.

3.2.1 Benefits

Class design provides for more sophisticated functionality—again, this is the argument for object-oriented design. When used well, this eases the development and maintenance burden. This style of analysis often helps in the creation of more sophisticated "correction" algorithms, where the advantages of object-oriented have been seen as valuable.

For example, DØ uses a format in which most reconstructed objects (electrons, tracks, jets, *etc.*) share a common implementation, and so can be used polymorphically. This has made it easier for physicists performing analysis to understand the different objects. It also makes it possible to write analysis software than can manipulate *any* of the physics objects, and thus makes sharing—and collaborative improvement—of code possible.

3.2.2 Drawbacks

Use of the additional power of classes generally requires CINT code, or when speed or reliability is required, writing C++. `ACLiC` makes the integration of such code into a ROOT session easier, but many physicists do not like writing analysis code in C++.

The `Draw` language is much less useful when dealing with real objects (as opposed to `structs`), because it does not support the use of general C++ constructions.

Code produced at an interpreter prompt (such as CINT or Python) often lacks design. No useful abstracts are identified, no increase in understandability of the code is obtained, and no correctness or performance benefits are realized. In short, when design is lacking, the advantages of object-oriented design are lost.

The use of real objects requires experiment-specific code; an "off-the-shelf" version of ROOT (or any analysis tool that does not store object code, as well as data) does not suffice.

Executables that use objects in ROOT tend to be much larger than those that use `struct`-like data, because the C++-based functionality of the objects is made available at the CINT prompt through the ROOT dictionaries and class method wrapper functions. This is a property of mixed-language programming, needed in this case to make function from a compiled language (C++) available from an interpreted language (CINT). This would not be necessary in a system that used an interpreted language throughout.

If the classes used in this object-based format are not the same as those used in the trigger and reconstruction code, "correction" algorithms developed for use with them can not be easily moved from the analysis environment to the production environment.

3.3 Summary

Both `struct`-like and object-like data formats are popular, and each will be wanted by many physicists. Their creation and use must be supported by the core infrastructure.

Why did physicists not use the event-data model and the framework provided by their experiments? Different users reported different reasons, sometimes reflecting differences between the event-data models of the experiments, and sometimes reflecting the different software development strategies favored by the physicist.

Some physicists prefer a "flat" (tuple-like) data model to a hierarchical model because it is easier to manipulate a flat model from the ROOT prompt. This is largely a reflection upon the strengths and weaknesses of the `Draw` language, which lacks sufficient facility for manipulating structures more complex than a (flat) tuple. This seemed to be commonplace at DØ, where the hierarchical nature of the data was very strong. Since the output of the DØ reconstruction program is typically stored in the *Event* as *collections* of objects, explicit C++ loops are needed to access each element (*e.g.* each track, each jet) in a collection. "Flattening" this hierarchy into a tuple makes the data available for use in the `Draw` language.

Some physicists found it hard to decode, or to understand, the hierarchical model. This was commonplace at CDF, where many of the algorithm groups, finding iterator interface of the CDF *Event* insufficiently convenient, wrote their own interfaces to event data. The end result was too many interfaces for some users to master. A few experts in the event-data structure wrote code which unwound the data of the event-data model into a flat tuple-like structure; non-experts then did not need to learn the many interfaces known only to the experts.

Some physicists found it inconvenient to read the descriptions of the available elements of the event data. Because the C++ classes were described in many header files, spread over many different subdirectories, learning about the available data was difficult. Unpacking the many classes into a few `structs` or classes made the amount to be learned smaller, and thus easier. At DØ, the classes written to one of the commonly-used object-based ROOT files are more closely related (*i.e.* they shared useful common base classes), than the classes used in reconstruction. This makes their interfaces and their use more uniform, and thus easier to learn. Perhaps because these analysis classes were defined later than were the analogous reconstruction classes, their design more closely reflected that which users (of this data format) wanted. Unfortunately, DØ never updated the classes used in reconstruction to take advantage of this improved understanding.

In some cases, is seems that the sort of separation of ideas that is useful for validation, maintenance and performance in the software has made it harder to understand the classes to those performing analysis. At least in part, this seems to be due to the difference in the point-of-view between the experts writing reconstruction code and the less-expert using the results of reconstruction. It seems that the classes optimized for best use in reconstruction are not best optimized for analysis use.

There is a tension between the short-term good and the long-term good. In the short term, the easiest way to add new information to the data is often to just put another bit of data into the collection of existing data—adding a new number to the tuple, with no additional organization needed. But after many such additions, the result is that the tuple (which may have been well-designed and coherent at first) grows incoherent and difficult to understand. Thoughts of long-term maintainability might have urged more design effort; but this is slower, and it is not always obvious what future uses will appear, and so it is not always obvious what the more maintainable design would be.

4 Data Handling

4.1 Handling Event Data

When the reconstruction output is not directly used to make final analysis tuples, and an intermediate tuple format is used instead, the process of performing corrections (and certifying data sets) becomes more complicated. Extra data handling is necessary, and this extra data handling is often not performed as part of the batch production system, but rather by individual algorithm groups or physics groups. The certification process delays the availability of the data for use by the physics groups, because this certification is *additional* certification beyond that required for

the reconstruction program itself. The certification process is time consuming. In some cases, this delay has amounted to several months. The programs used to process the intermediate tuple format amount to another framework. Since this framework is often under-developed, and lacks the features of the experiment's reconstruction framework, its use is often more difficult. For example, automatic of handling of metadata is sometime non-existent, and so tasks performed automatically by the reconstruction program are done "by hand", and in a less reproducible and certain manner, by the physicists running the reprocessing programs.

Obviously, saving data in multiple formats requires more storage, and generating those multiple formats requires more processing power, all of which increases the data handling burden on an experiment. However, in both CDF and DØ, this storage space and processing power is used, in efforts centralized to differing degrees, because physicists will produce data sample which fit their perceived needs whether or not such samples are provided by their experiment's production system. CMS will benefit from making sure its event-data model, and metadata model, help ease the burden.

4.2 Handling Metadata

In general, those using `struct`-style data do so because they prefer *simplicity* in their analysis software; those using object-style data d o so because they prefer *power*. The metadata handling for each of these styles of use should be made to fit the different styles.

What metadata access to users of `struct`-style data want? Because they want to be able to work without experiment-specific libraries, it would seem sensible to store the most important metadata in the event data itself. For example,

- the version numbers of critical algorithms,
- the most important parameters of critical algorithms, and
- the version number of the production program.

In addition, it is possibly valuable to have some compressed identifiers in the event data which can be used (possibly outside the analysis system) to look up the full provenance information elsewhere. That "elsewhere" might be in another file (perhaps in the same file format as the event data themselves), or it might be in a more formal "database" (anything from a local SQLite or Berkeley DB file, to a remote PostgreSQL, MySQL or Oracle database).

Those using object-style data may be interested in more options for programmatic access to the metadata. For such users, it would be beneficial for the metadata model of the analysis system to be as similar as possible to the metadata model of the trigger and reconstruction system. After all, the major reason for the production system's metadata model is to provide the information necessary in analysis.

5 Conclusion

In this document, we have described features of the infrastructure software of two current experiments. In hindsight, we can identify several flaws in their systems. We have described some of them in this document. Figure 1 summarizes the flow of data described in this document.

Analysis and design shortcomings in the reconstruction frameworks led to difficulties in the analysis stage. It is important to note that it seems inevitable that some fraction of physicists will avoid the reconstruction and analysis framework. By recognizing this early, one can prepare for it in the core infrastructure, rather than requiring modifications at a later date. One can also plan for adequate handling of metadata, for reproducibility purposes; this is often lacking in the

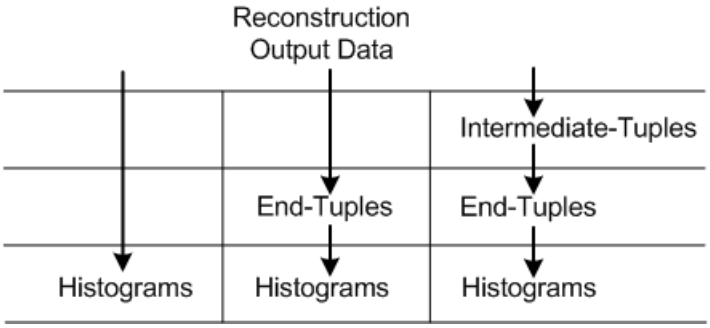

Figure 1: The three main data flows appearing in physics analysis.

ad hoc "intermediate tuple" translation. Such design can also reduce the effort required for a physicist to begin a new analysis. It can also reduce the redundancy of data. Some redundancy of data is beneficial, *e.g.* local copy of "final sample" for thesis analysis, because the various trade-offs among different physicists' perceptions of benefits can not be "solved" once for all physicists.

A goal of the design should be the elimination of the "intermediate tuple" format. Creation, storage, and handling of this intermediate tuple is an unnecessary burden.